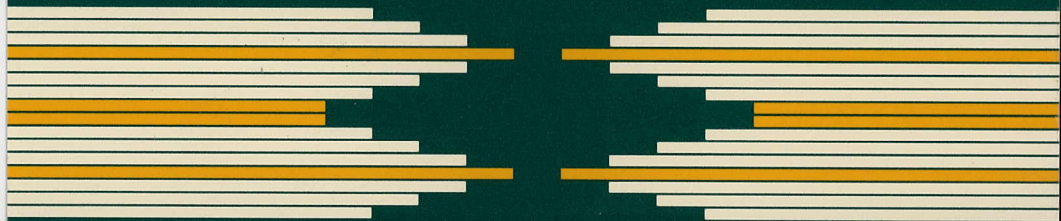


MICROSOFTTM
APPLESOFT[®]

COMPILER
System



For Apple II

Microsoft® Applesoft® Compiler

User's Manual

HotLINE: 206 828-8089

Registration No: 150533 4 JAN 83

VSN: 201

Microsoft Corporation

Product Support Dep.

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the software in this package on any medium for any purpose other than personal use.

©Copyright Microsoft Corporation, 1981, 1982

Comments about this documentation may be sent to:

Microsoft Corporation
Microsoft Building
10700 Northup Way
Bellevue, WA 98004
(206) 828-8080

Microsoft is a registered trademark and RAMCard is a trademark of Microsoft Corporation.

Apple and Apple Language System are registered trademarks of Apple Computer, Inc.

Catalog No: 2222
Part No: 22F22B
Doc. No: 8106-200-00

Contents

1	Introduction	1
	System Requirements	2
	How to Use this Manual	3
	Contents of the Microsoft	
	Applesoft Compiler Package	5
	Runtime License Requirements	5
	Resources for Learning Applesoft	6
2	Demonstration Run	7
3	Introduction to Compilation	11
	Vocabulary	11
	Compilation vs. Interpretation	12
	The Program Development Process	15
4	Debugging with the Applesoft Interpreter	17
	Creating a Source Program	17
	Running a Program with Applesoft	17
5	Compilation	19
	Options	19
	Terminating Compilation	24
	Compiling Large Programs	24
6	Executing a Compiled Program	27

7	A Compiler/Interpreter Language Comparison	31
	Statements Not Implemented	31
	Features Supported with Limitations	32
	Other Language Differences	35
	Operational Differences	37
8	Language Enhancements	45
	Integer Arithmetic	45
	Chain with COMMON	52
9	How the Compiler Works	61
	PASS0, PASS1, and PASS2	61
	Syntax Analysis	63
	Code Generation	65
	Special Techniques	65
10	Error Messages and Debugging	69
	Compiletime Error Messages	69
	Runtime Error Messages	71
	Sources of Common Problems	72

Appendices

A	Moving Binary Files with the ADR Utility	77
B	Creating a Turnkey Disk	83
C	Notes on Applesoft	85
D	Runtime Memory Map	89
E	Zero Page Usage	91

Index	93
--------------	-----------

Chapter 1

Introduction

The Microsoft® Applesoft® Compiler is designed to complement the Applesoft interpreter, to extend the Applesoft language, and to enhance execution of Applesoft programs. The Applesoft interpreter itself was designed by Microsoft Corporation and later modified by Apple® Computer, Inc. The interpreter/compiler combination is the ideal Applesoft program development tool, since programs can be quickly entered and debugged with the interpreter, then optimized for speed with the compiler.

The compiler supports the Applesoft language with only a few exceptions. Therefore, most of the programs already written in Applesoft for the Apple® II can be compiled with little or no change.

Other major benefits provided by the Microsoft Applesoft Compiler are:

Increased Execution Speed

Applesoft programs compiled with the compiler normally run from two to twenty times faster than the same programs run under the interpreter.

Interprogram Communication

Programs can communicate with each other by using COMMON variables.

True Integer Arithmetic

Unlike the Applesoft interpreter, the compiler can perform true integer arithmetic. Integer arithmetic can greatly increase execution speed.

Applesoft Compiler

Source Code Security

The Microsoft Applesoft Compiler creates machine-language equivalents of Applesoft BASIC programs. This machine-language file is all that need be distributed when a commercial application is sold. Therefore, the Applesoft program (called a "source" program) is protected from copying or plagiarism.

Disk-based Compilation

Unlike other Applesoft compilers that create the machine-language version of the program in memory, the Microsoft Applesoft Compiler writes out the machine-language program to disk as it compiles. This allows the Microsoft Applesoft Compiler to compile programs of virtually any size.

These benefits are important for speed-critical applications, such as graphics, and for applications in which a large system of programs must be supported by a main menu. The Microsoft Applesoft Compiler is also outstanding for commercial applications sold in a competitive marketplace that require source code security.

The Microsoft Applesoft Compiler is particularly good for programs that are otherwise too large to fit in memory. By separating such programs into parts and communicating between them with COMMON variables, large systems of communicating programs can be created. The compiler is an example of such a large system, since it was separated into parts and used to compile itself. This gives an indication of the impressive power of the compiler as a programming tool.

System Requirements

The Microsoft Applesoft Compiler requires an Apple II Plus or an Apple II computer with the Applesoft firmware card installed, 48K RAM and a minimum of one disk drive (16 sector-DOS 3.3). The compiler supports, but does not require, the Microsoft® RAMCard™ or Apple® Language System.

How to Use this Manual

The first three chapters of this manual are designed for users who are unfamiliar with the compiler as a programming tool. Successive chapters give a thorough technical description of the Microsoft Applesoft Compiler and its use. This manual is organized as follows:

- | | | |
|-----------|---------------------------------------|---|
| Chapter 1 | <i>Introduction</i> | Provides brief descriptions of the contents of the Microsoft Applesoft Compiler package, and lists resources for learning Applesoft programming. |
| Chapter 2 | <i>Demonstration Run</i> | Explains the compilation and execution of a demonstration program. |
| Chapter 3 | <i>Introduction to Compilation</i> | Introduces the vocabulary associated with compilers, compares interpretation and compilation, and briefly describes program development with the compiler. |
| Chapter 4 | <i>Debugging with the Interpreter</i> | Describes how to debug the source file with the Applesoft interpreter before compiling the same source. (Chapter 7 describes differences between the Microsoft Applesoft Compiler and the Applesoft interpreter.) |
| Chapter 5 | <i>Compilation</i> | Describes in detail the use of the compiler, including the various compiler options. |
| Chapter 6 | <i>Executing a Compiled Program</i> | Describes how to run a compiled object file. |

Applesoft Compiler

- Chapter 7 *Compiler/Interpreter
Language Comparison*
Describes all of the language, operational, and other differences between the compiler and the Applesoft interpreter. This chapter should be read before attempting to compile any programs with the compiler.
- Chapter 8 *Language Enhancements*
Describes the compiler's extensions to Applesoft.
- Chapter 9 *How the Compiler Works*
Describes the inner workings of the compiler.
- Chapter 10 *Error Messages and Debugging*
Describes each error message and discusses common problems.

Appendices include:

- A. *Moving Binary Files with the ADR Utility*—Describes how to BLOAD and BSAVE binary files with the ADR utility.
- B. *Creating a Turnkey Disk*—Describes how to create a turnkey disk for compiled programs.
- C. *Notes on Applesoft*—Discusses unusual Applesoft features.
- D. *Runtime Memory Map*—Shows memory usage by compiled programs.
- E. *Zero Page Usage*—Shows zero page locations used by compiled programs.

This manual assumes that the user has a working knowledge of the Applesoft language. For additional information on Applesoft programming, refer to page 6, "Resources For Learning Applesoft."

Contents of the Microsoft Applesoft Compiler Package

The Microsoft Applesoft Compiler package includes this manual and one disk, which contains the following files:

1. APCOM—The Microsoft Applesoft Compiler.
2. PASS0, PASS1, PASS2—Internal components of the compiler.
3. RUNTIME—Library of machine-language routines used by compiler programs.
4. ADR—Utility for binary files.
5. CREATE ADR—Utility for creating ADR on other disks.
6. BALL—Demonstration program.

Important

Language, operational, and other differences between the Microsoft Applesoft Compiler and the Applesoft interpreter are described in Chapter 7. Read Chapter 7 before attempting to compile any programs with the compiler.

Runtime License Requirements

The Microsoft Applesoft Compiler is authorized for single computer use only, and only by the registered owner. No royalties are required for application programs created with the compiler. Applications created with the compiler may be distributed with the runtime module. However, the message

"PORTIONS COPYRIGHTED BY MICROSOFT CORP., 1981"

must appear on the media and documentation.

Resources for Learning Applesoft

This manual provides complete instructions for using the compiler. However, it does not provide tutorial material for learning Applesoft. The following texts are good sources for this information:

Albrecht, Robert L., Finkel, Leroy, and Brown, Jerry. *BASIC*. John Wiley & Sons. 1973.

Apple Computer Inc. *Applesoft II BASIC Programming Reference Manual*. 1978.

Apple Computer Inc. *The Applesoft Tutorial*. 1979.

Coan, James S. *Basic BASIC*. Hayden. 1970.

Dwyer, Thomas A. and Critchfield, Margot. *BASIC and the Personal Computer*. Addison-Wesley. 1978.

Chapter 2

Demonstration Run

This chapter provides step-by-step instructions for using the compiler. We strongly recommend that you perform the demonstration run *before* compiling any other programs, since the demonstration run gives you a working overview of the compilation process. If you enter commands exactly as described in this chapter, you should have a successful session with the compiler.

Important

Before beginning this demonstration run, make a backup copy of the Microsoft Applesoft Compiler disk. Store the master disk in a safe place and work with the backup copy.

The Microsoft Applesoft Compiler is simple to use. To invoke the compiler, first boot up your disk and then type:

```
JRUN APCOM
```

(Note that the prompt is in boldface type.)

Next, a few simple questions must be answered to begin the actual compilation process. The first two prompts ask you for the names of the source and object files:

```
SOURCE FILE? BALL
```

```
(DEFAULT BALL.OBJ)? <RETURN>
```

Applesoft Compiler

The source file is an Applesoft program named BALL that already exists on disk. The object file is the machine-language binary file that is created by the compiler. The object file name defaults to the original file name with the extension .OBJ added, so the object file produced for the BALL program is BALL.OBJ. The default is specified by entering <RETURN>.

The source file is assumed to be located on the same disk as the compiler, unless you specify otherwise. The object file defaults to the same disk that the source is on. Different slots or drives can be specified using the normal ,S<slotnumber> and ,D<drivenumber> syntax. Compilation is usually faster if only one drive is used. Disk commands can be executed by typing <CTRL-D>, followed by the command and <RETURN>.

The next two prompts ask you whether you want default values for all other compilation options. Since most compilations are performed with the same set of options, you should enter <RETURN> after each prompt to specify the default values:

MEMORY USAGE:
DEFAULT CONFIGURATION? <RETURN>

OPTIONS:
DEFAULT CONFIGURATION? <RETURN>

If you had refused the default configurations above, you would need to explicitly specify the values of several compilation options. These options are explained in Chapter 5. The actual compilation process starts without further input, since you have specified the defaults above.

When compilation begins, the disk is accessed almost constantly to either read the source file or to write the object file. The compiler lists the source program on your console as it is being compiled and generates appropriate messages if it encounters any errors. When the source stops listing, the first part of compilation is finished, and the compiler prints:

*****BEGINNING PASS2

The second part of compilation also uses the disk extensively. To indicate that it is still compiling, the compiler prints a period on the screen every few seconds. When it is finished, the compiler prints:

*****CODE GENERATION COMPLETE

At this point, the actual compilation process is complete. To receive a listing of compilation information, answer "Y" or "YES" to the next prompt:

**COMPILATION INFORMATION AND LINE NUMBER
REFERENCE TABLE? YES**

This input also accepts <CTRL-D> disk commands. If you want to list the compilation information on a printer, first turn on your printer by entering:

<CTRL-D> PR#<printerslot>

The compiler prints out the desired information, displays the following message, and then reenters the interpreter:

*****COMPILATION COMPLETE

]

The increase in the BALL program's execution speed is quite apparent when compared to the same program running under the interpreter. Compare speeds by first running the interpreted program:

]RUN BALL

Next, execute the compiled program by entering the following DOS commands:

]BLOAD RUNTIME
]BRUN BALL.OBJ

Note that the RUNTIME library must be BLOADED in memory before BALL.OBJ can be BRUN.

You have now successfully completed the demonstration run. Be sure to read Chapter 7 before attempting to compile other Applesoft programs.

Chapter 3

Introduction to Compilation

This chapter introduces the vocabulary of compilation, compares compilation to interpretation, and describes the program development process for compiled Applesoft programs. These three topics serve as an introduction to compilation.

Vocabulary

Although this manual attempts to keep technical language to a minimum, the following terms must be understood:

- | | |
|--------------------|---|
| Source File | The Applesoft program is commonly called a source file because it is the source from which an equivalent machine-language file is created. The source file is the input file to the compiler. CATALOG lists the names of Applesoft source files, with the letter "A" preceding the size of each file. |
| Object File | The Microsoft Applesoft Compiler translates source files into machine-language object files. The object file is the output file created by the compiler. The object file is an executable binary file that is the machine-language equivalent of the source. CATALOG lists the names of object files, with the letter "B" preceding the size of the file. |

Applesoft Compiler

Compiletime	The time during which the compiler is translating a source file into an object file.
Runtime	The time during which a compiled program is executing. By convention, runtime refers to the execution time of a compiled program, and not to the execution time of the compiler.
Runtime Library	A collection of machine-language routines that are used by compiled object programs. These routines all reside in the file named RUNTIME. RUNTIME must be loaded into memory before an object file can be executed.

Compilation vs. Interpretation

Since the microprocessor in the Apple computer can execute only its own machine instructions, it does not execute Applesoft program statements directly. Instead, statements must be simulated by machine-language routines that perform the operations specified by each BASIC statement.

Compilers and interpreters approach this translation problem differently. This difference in method is demonstrated in the following analogy.

Suppose you wish to build a stereo from a kit. Unfortunately, you find that the provided instructions are written in Japanese, a language that you do not know.

One way to approach this problem is to translate the instructions with a Japanese-English dictionary and then perform each instruction one-by-one. This process parallels the interpretation of Applesoft statements by the Applesoft interpreter. Your construction of a stereo kit is analogous to the Apple computer's execution of a program.

Inefficiencies can arise in this process, especially if you fail to write down the translated instructions as they are carried out. Suppose, for example, that halfway through the construction process you translate an instruction that says:

Go back to Instruction 14. Then repeat the preceding steps for the second speaker.

One problem is immediately apparent: you can't even find Instruction 14 without scanning the instructions from the start for the Japanese characters for fourteen. You then face the time-consuming task of retranslating each instruction so the second speaker can be assembled precisely like the first. Likewise, the Applesoft interpreter must repeatedly translate each statement executed inside a FOR/NEXT loop.

An alternative approach to constructing the stereo is to sit down with pencil and paper and translate the entire instruction sheet into English. When you are finished, you have new instructions written in English. These English instructions correspond to the object file created by the compiler.

The actual assembly of the kit can then proceed without any further translation, and even without any further need for the original Japanese text. When you finally sit down to build your speakers, you can construct them very quickly. Similarly, a compiled BASIC program does not require the original BASIC source and runs very quickly, in comparison to the interpreted version of the same program.

This analogy gives you a feel for the difference between interpretation and compilation. The following paragraphs discuss interpretation and compilation more technically and more directly.

Interpretation

The interpreter translates Applesoft source statements line-by-line at runtime. Each time the interpreter executes an Applesoft statement, it must analyze the statement, check for errors, and call machine-language routines that perform the desired function.

When statements must be executed repeatedly, as must those within a FOR/NEXT loop, the translation process must be repeated each time the statement is executed.

In addition, BASIC line numbers are stored in a list. GOTOs and GOSUBs force the interpreter to search this list to find the desired line. This search is slow when the line is near the end of a long program.

The interpreter keeps track of variables by using a list, too. When it encounters a reference to a variable, the interpreter searches from the beginning of the list to find the desired variable. If the variable is not present in the list, the interpreter creates a new entry for it. This procedure also slows interpreted programs.

Compilation

A compiler, on the other hand, takes a source program and translates it into a machine-language object file. This object file consists of a large number of machine-language CALLs to routines in the runtime library and to routines in the Applesoft interpreter. By calling routines in the Applesoft interpreter, the Microsoft Applesoft Compiler assures close language compatibility with the interpreter.

Unlike the interpreter, the compiler analyzes all statements *before* runtime. In addition, absolute memory addresses are provided for variables and program lines. These addresses eliminate the list searching that occurs while an interpreted program executes.

The compiler, unlike the interpreter, implements true integer arithmetic and integer loop variables in FOR/NEXT loops. In comparison, the Applesoft interpreter converts all integers to real numbers before operating on them. These conversions make interpreted integer arithmetic relatively inefficient. In addition, the interpreter forbids use of integers as loop control variables in FOR/NEXT loops.

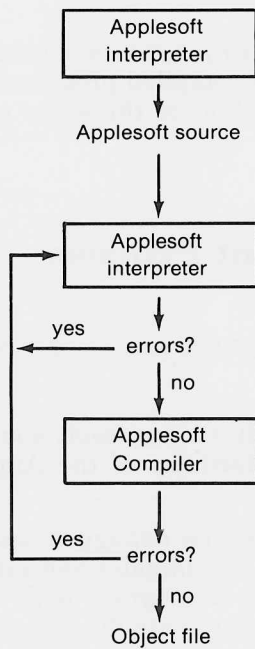
These factors combine to make compiled programs considerably faster. In most cases, execution of compiled programs is two to twenty times faster than execution of the same programs under the interpreter.

The Program Development Process

This discussion of the program development process is keyed to Figure 1.

1. The best way to create and edit an Applesoft source program is to use the editing facilities of the Applesoft interpreter.
2. The source program should then be debugged using the Applesoft interpreter. Since the compiler and interpreter are very similar, running a program with the interpreter provides a much quicker syntactic and semantic check than does compiling the same program.
3. Next, the source program is compiled. If compilation is successful, the compiler produces an object file.
4. Finally, the object file is executed as a machine-language program. If the compiler detects errors, or if errors show up while executing the compiled program, the process starts over at step 2.

Applesoft Compiler



1. Create and edit Applesoft source.

2. RUN and debug source with the interpreter.

3. Compile source, creating a binary object file.

4. Execute compiled object file.

Figure 1. The Program Development Process

Chapter 4

Debugging with the Applesoft Interpreter

Debugging a program intended for compilation is a two-step process that involves:

1. Creating the source program.
2. Running the program under the interpreter to check for errors.

These two steps are described below.

Creating a Source Program

Creating an Applesoft source program requires the use of the editor available within Applesoft. Programs are created by simply entering Applesoft statements from within Applesoft. Once a program has been created, it can be saved to disk with SAVE. The compiler can only compile Applesoft disk files. For more information on creating source files, see the *Applesoft II BASIC Programming Reference Manual* and the DOS Version 3.3 manual.

Running a Program with Applesoft

Programs should be debugged before compilation with the Applesoft interpreter. It may be necessary to debug the program with the compiler if the program to be compiled uses Microsoft Applesoft Compiler features that are not available in the interpreter. See Chapter 8 for other debugging suggestions.

Applesoft Compiler

The Microsoft Applesoft Compiler is highly compatible with the Applesoft interpreter. This compatibility allows the Applesoft interpreter to function as the primary debugging tool. The interpreter provides much better debugging facilities than does a compiler, since it includes features such as TRACE. With the interpreter, programs can also be halted and the values of variables examined, with immediate mode PRINTing of their values. In addition, an interpreted program can be modified without having to repeat the more lengthy compilation process.

There are some drawbacks to debugging with the interpreter. Statements that are only executed under special circumstances may never be examined. Also, the interpreter halts execution when it encounters the first error in a program.

On the other hand, the compiler examines every statement in a program, and can continue the compilation even if it encounters errors.

In general, compiling a program is an effective way to check for syntax errors; however, program logic errors are more easily tracked with the interpreter.

Chapter 5

Compilation

This chapter discusses the more technical aspects of compilation. It includes the following sections:

1. Options
2. Terminating Compilation
3. Compiling Large Programs

Note

If a compiled program does not run correctly, see Chapter 10 for possible solutions. Chapter 7 also provides information about possible problems and their solutions.

Options

The demonstration run showed only the most basic type of compilation. The Microsoft Applesoft Compiler includes several options that can be used to control compilation more closely. These options control memory allocation and compilation. To explicitly specify the values for these options, simply answer "NO" when the compiler offers the default values, then enter a value.

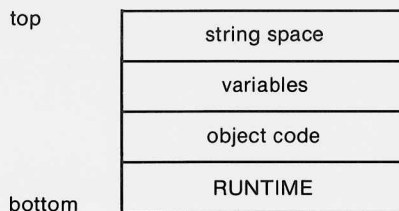
Memory Usage

The memory used at runtime by the compiled code is divided into three areas:

1. RUNTIME Library
2. Object Program
3. Variables

The compiler allows the location for each of these separate blocks to be specified individually. The memory allocation features can be used to protect machine-language programs, and to shape tables, the HIRES screens, or any other important part of memory.

The default allocation order of the blocks is: library, program, variables. The library is allocated lowest in memory, and the program and variables follow. The library begins at location 2051, or \$803 (with the dollar sign (\$) indicating a hexadecimal value). The default configuration for memory is:



Alternate addresses for the blocks are simple to specify. The new location for the library is entered as a number and defaults to \$803. Addresses can be specified in either hexadecimal or decimal form. Hex addresses must be preceded by a dollar sign (\$).

The library must be BLOADED before a compiled program can be run. By default, the library is loaded at \$803. When a program is compiled to expect the library at a different

address, the library must be loaded in at the correct address by using the "A" option with the BLOAD command. See Appendix A for more information about loading and saving the binary object and RUNTIME files.

The beginning address for the object code may be specified with:

1. The word HGR1
2. The word HGR2
3. A decimal or hex number
4. <RETURN>

HGR1 and HGR2 simply set the beginning of the program above the appropriate HIRES screen. The 4K RUNTIME library defaults to the space below the first HIRES screen. This default library location is suggested for programs that use HIRES graphics. An absolute decimal or hex address may also be specified, but care must be taken when doing so. Beware of overlapping memory allocations. Typing <RETURN> causes the beginning of the object code to default to the end of the library.

Variable space may be specified explicitly or allowed to default. The beginning of variable space defaults to the end of the object code.

Compiled programs use the normal HIMEM pointer to determine the top of available string space. Strings grow downward from HIMEM. The bottom of available string space is set so that the block that is highest in memory is protected. Therefore, the normal default order (RUNTIME library, object program, variables) sets the bottom of string storage to the end of variable space. Specifying another block to reside highest in memory sets the bottom of string storage to the end of that block.

Compilation Options

There are five compilation options that can be specified before the compilation process begins.

These options have the following defaults:

Compilation Option	Default
Compilation Listing	Yes
Pause on errors	Yes
Integer	Yes
Integer constants	Yes
RESUME/Debug code	No

Answering "No" or "N" to the default option prompt provides a chance to turn each of these options on or off.

Compilation Listing Option

The compiler normally lists the source file. Turning the listing option off suppresses the listing. Errors, warnings, and special messages are printed as usual.

Pause on Errors Option

Errors normally halt compilation and allow the user to abort or continue compilation. Turning the pause option off suppresses the pause after any error messages are printed.

Integer Arithmetic Option

The compiler includes a full integer arithmetic package. True integer arithmetic allows operations on integers to be performed in about half the normal time. Including this option substantially increases the speed of programs that use integers, but there are some limitations. See Chapter 8 for more information on the integer arithmetic package.

Integer Constants Option

Constants in a compiled program can be treated as integers or floating-point numbers. Selecting the integer constants option allows constants that are used as integers to be stored in integer format. If a constant is also needed in floating-point form, the compiler includes both forms with the compiled code. Conversion of constants at runtime is totally eliminated. Including the integer constants option increases the speed of programs where constants are accessed as integers.

Integer constants take up two bytes in the object file; the floating-point representation requires five. Including integer constants can slightly increase the size of the object code if both real and integer versions are needed, but it can also shorten the code if only the integer representation is required. The integer constants option should normally be left on. See page 24, "Compiling Large Programs," for more information on handling any problems with longer programs.

RESUME/Debug Code Option

Turning on the RESUME/Debug code option causes code handling the RESUME statement to be included in the object program. The RESUME statement in Applesoft allows an error-trapping routine to resume execution at the beginning of the statement that caused the error. Unlike some Applesoft compilers, the Microsoft Applesoft Compiler fully supports ONERR GOTO. The compiled version of ONERR GOTO traps all runtime errors, including those that occur within routines from the Applesoft interpreter.

Including the RESUME/Debug option requires the compiler to generate extra code at the beginning of each statement that may generate an error. Selecting the RESUME/Debug code option causes the object code to be larger and somewhat slower.

The advantage of using the RESUME/Debug code option is that runtime error messages include the object code address. Normally, only some of the errors generated by the runtime

library include an object code address. The RESUME/Debug option can be useful for debugging with the compiler. However, including it does decrease the speed and increase the length of the compiled code. The RESUME/Debug option should be left off unless it is absolutely needed. If the option is off the compiler ignores all RESUME statements.

Terminating Compilation

Ordinarily, <RESET> would be used to terminate compilation of a program. However, since the compiler often accesses the disk, using <RESET> is inadvisable. Instead, use <CTRL-C> to terminate compilation. This works because the compiler occasionally checks to see if a <CTRL-C> has been typed, and terminates compilation if it has. The compiler can also be halted by typing <CTRL-C> as the first character of an input response, but this does not correctly terminate compilation.

Since stopping compilation leaves the object file incomplete, the compiler deletes the object file if compilation is aborted. The compiler modifies DOS, so exiting by using <RESET> or typing <CTRL-C> in an input leaves DOS in its modified state. DOS must then be rebooted. Typing <CTRL-C> outside an input is the only way to correctly terminate compilation. Normal exiting of the compiler restores DOS to its normal state.

Compiling Large Programs

Features such as compact object code, disk-based design, and COMMON variables make the Microsoft Applesoft Compiler an outstanding compiler for large programs. Because of the compiler's minimal object code expansion, most large programs can be compiled without modification. However, if a program is simply too large to be compiled or to fit into the available memory, special measures must be taken. The sections that follow describe possible solutions.

Reducing Symbol Table Space

The compiler's symbol table must store information about variables, functions, constants, line number references, and COMMON assignments.

The symbol table is examined before each new entry is made. This examination prevents duplication of information. The first use of a variable requires the compiler to create a new entry, but later references do not require additional space. Similarly, multiple references to a single line number do not require multiple entries.

Functions require four symbol table locations. Line number references require three. Simple variables require five locations, and array variables require six plus one for each dimension. COMMON variables require an additional two locations per variable. In practice, it is difficult to eliminate functions, line number references, or variables, but it may be possible.

Long programs that exhaust symbol table space yield a "SYMBOL TABLE FULL" error during compilation. There are several ways to correct this problem.

The simplest way is to turn off the integer constants option. With the option on, constants are initially stored as integers. If the constant is later needed as a floating-point number, it is converted and entered into the symbol table as a floating-point constant as well. The initial integer entry takes five locations. If the additional floating-point entry is required, it takes up eight locations.

With the integer constants option off, constants are stored only in the floating-point form. Therefore, turning the option off saves five locations for every constant that is referenced as a floating-point value. Although this savings is usually insignificant, it may be important for long programs with many constants.

Turning the integer constants option off also slows down the object code slightly. The option should be left on whenever possible. If the "SYMBOL TABLE FULL" error persists, a program can often be separated into two parts, as described below.

Separating a Program into Parts

When the object code for a program does not fit in the available memory, or when a program requires too much symbol table space, the program can often be separated into two smaller programs. Programs that spend most of their time in one section of code, then move on to another section and do not return, are good candidates for this technique. A program of this type can quite easily be broken into two smaller programs. The first small program performs the first part of the process, then passes any needed values on to the second program.

Programs without a natural division present more of a problem. An artificial division can often be created with this type of program. In some cases, the separated programs may have to run each other alternately. It is preferable to find a division where alternate execution is not required, since running programs alternately from disk is very slow.

Since most programs that are separated into parts need to pass values from one program to another, the usual procedure is to pass the needed values in a disk file. One program writes the information out to disk, and a second reads it in. This is a workable solution, but it is slow when a large amount of information needs to be passed.

The Microsoft Applesoft Compiler is designed to simplify compiling large programs. The compiler allows you to pass the needed values in COMMON. The variables are simply declared in COMMON statements in both programs, and the compiler allocates storage so that saving the values on disk is unnecessary. This is the technique that the compiler uses to communicate between its three parts: PASS0, PASS1, and PASS2. See Chapter 8 for more information about the compiler's powerful COMMON features.

Chapter 6

Executing a Compiled Program

The Microsoft Applesoft Compiler is designed to implement the Applesoft language as closely as possible. Executing a compiled program performs the same BASIC statements and functions as executing the source program using the interpreter. However, the process of loading and running a compiled program is different from running an interpreted program, because the object file is a machine-language program stored as a binary file.

Differences between the compiler and the interpreter and other related topics are discussed below:

1. Program Storage

Interpreted programs are stored as Applesoft files. Applesoft files are indicated by an "A" in the disk CATALOG. Interpreted programs are executed by typing `RUN <filename>`. In contrast, compiled programs are machine-language files stored in binary format. Binary files are indicated by a "B" in the CATALOG.

Since compiled programs are not Applesoft files, attempting to `RUN` a compiled program gives a "FILE TYPE MISMATCH" error. Instead, compiled files are executed by typing:

`BRUN <filename>`

The "B" prefix to the `RUN` command indicates a binary file. The `RUNTIME` library must already be in memory when a program is `BRUN`. The library need not be reloaded if the `RUNTIME` library is still in memory from a previous program run and is located where the next program expects it.

Applesoft Compiler

The normal sequence for executing a compiled program is:

```
BLOAD RUNTIME  
BRUN <filename>
```

Compiled programs can only be executed when the Applesoft interpreter is in memory. Compiled programs will not work with Integer BASIC.

2. Use of Ampersand (&)

Once the compiled program has been loaded and executed, it can be reexecuted by typing an ampersand followed by a <RETURN>. The compiled program sets the ampersand vector to point to the beginning of the object code when it executes, so the ampersand vector can be used as long as the program is the last program run. Using the ampersand is much more convenient than CALLing the code explicitly, since CALLing the code requires knowing the address of the beginning of the object code.

3. Halting Execution of a Compiled Program

<CTRL-C> does not function during execution of a compiled program, since the compiled code executes directly and is not under the supervision of the interpreter. Unless the object program is explicitly looking for input, any characters typed are ignored.

However, typing <CTRL-C> as a response to an INPUT statement functions exactly the same as in the interpreter. This means that compiled programs must be interrupted by using <RESET>. <RESET> does not correctly reinitialize the Applesoft interpreter, so a NEW command must be used after interrupting a compiled program.

4. The NEW Command

NEW causes the interpreter to reset pointers, but not to clear the program space. Therefore, a compiled program can be safely reexecuted if no program lines have been typed in and stored into the program space.

5. Immediate Commands

The compiled code does not maintain a variable list, so the interpreter cannot find the values of variables used in a compiled program. Executing a compiled program that uses the variable A and then typing the immediate command PRINT A returns a value unrelated to the variable in the compiled program.

Chapter 7

Compiler/Interpreter Language Comparison

The Microsoft Applesoft Compiler produces compact and efficient code that simulates the Applesoft interpreter in very accurate detail. This chapter describes any differences that must be taken into account when compiling programs. If a compiled program does not run correctly, see Chapter 10 for more information.

Statements Not Implemented

The very nature of compilation makes supporting some features of the interpreter impractical, since the source file is unavailable while the object code produced by the compiler is executing. Therefore, Applesoft statements that depend on the source file (such as LIST and DEL) are not available with the Microsoft Applesoft Compiler. The cassette I/O features of Applesoft have also been removed. Most other features of Applesoft are implemented without change.

The following Applesoft statements are not included in the compiler:

CONT	RECALL
DEL	SAVE
LIST	SHLOAD
LOAD	STORE
LOMEM:	TRACE
NOTRACE	&

Some other Applesoft statements and features must be used differently in compiled programs. These differences are described in the following sections.

Features Supported with Limitations

The following Applesoft features are supported with some limitations:

DEF FN
DIM
<CTRL-C>

The differences between the compiled and interpreted versions of these statements are described below.

User DEFined Functions

Both the Microsoft Applesoft Compiler and the Applesoft interpreter allow the definition of single-argument, real-valued arithmetic functions with the DEF FN statement. In addition, the interpreter allows functions to be redefined with a later DEF FN statement using the same function name. The compiler *does not* support function redefinition.

In the interpreter, a DEF FN does not define a function until the DEF FN statement is actually executed at runtime. The compiler, on the other hand, scans all function definitions at compiletime. Therefore, function definitions can be located anywhere within the source file, and functions are defined from the beginning to the end of program execution. The source file cannot contain more than one definition for a given function, even if the definitions are identical.

The compiler's treatment of user-defined functions prevents "?UNDEF'D FUNCTION" error messages at runtime. All user-defined function references are matched with the corresponding definition at compiletime. References to undefined functions are detected and flagged as errors during compilation.

The DIMension Statement

The interpreter provides three methods of dimensioning an array:

1. Explicit Constant Dimensioning

Executing a DIMension statement in which the specified dimensions are constants sets aside the same amount of storage for the array each time the program is run.

2. Explicit Dynamic Dimensioning

Executing a DIMension statement in which the specified dimensions are arithmetic expressions sets aside space for the array, depending on the computed value of the expressions.

3. Default Dimensioning

If an array reference is encountered before a DIMension statement, the array is given the default maximum value of 10 for each dimension of the array. Accessing an element of a three-dimensional array before dimensioning the array produces the default dimensions (10,10,10). Applesoft allows the use of 0 as an array subscript, so an array dimensioned to 10 actually has 11 elements (0-10).

The compiler *does not* support dynamic dimensioning of arrays. Any DIM statements in the program must use *integer* constants, not floating-point values or arithmetic expressions. DIMension statements can be located anywhere within the file, and they need not precede the first reference to the array. Arrays can be dimensioned more than once, but the dimensions specified must be identical. The compiler's added INTEGER and COMMON statements can also dimension arrays. See Chapter 8 for more information on INTEGER and COMMON. An array that is referenced and not dimensioned receives a default dimension of 10 for each subscript.

References to an array within a DIMension statement or arithmetic expression must be consistent in the number of dimensions used throughout the file. An error is generated if an array is referenced with a number of subscripts that differs from the number first used.

The lack of dynamic array dimensioning can be overcome by dimensioning arrays to the largest possible value allowed within the constraints of available memory. Since keeping dimensions small has little other advantage than conserving memory, all available memory can be assigned to arrays if the maximum subscripts needed for the arrays are unknown.

Use of <CTRL-C> to Halt a Compiled Program

The Applesoft interpreter allows the user to interrupt execution of a program by typing a <CTRL-C>. In addition, typing <CTRL-C> followed by a <RETURN> during INPUT causes the program to halt.

Compiled code does not check for <CTRL-C> during execution. A compiled program can be halted only by using the <RESET> key. However, <RESET> interrupts cannot be trapped by compiled programs. As a result, <RESET> does not properly terminate compiled programs, and a NEW command must be executed to re-initialize the interpreter. See Chapter 6 for more information on the use of <RESET>.

While the compiled code does not check for <CTRL-C> during execution, it *does* support the use of <CTRL-C> during an INPUT statement. Typing <CTRL-C> followed by a <RETURN> in response to an INPUT prompt causes program termination and a "BREAK IN ####" message. The compiled INPUT functions the same as the interpreted INPUT.

If necessary, a compiled program can simulate the interpreter's handling of <CTRL-C> during execution by periodically checking the keyboard strobe. See the *Applesoft II BASIC Programming Reference Manual* for more information about the keyboard strobe.

Other Language Differences

A few Applesoft statements have been modified to return reasonable results under a wider variety of conditions. These are described below.

IF/THEN Using Strings

The Applesoft interpreter was not designed to allow the IF/THEN statement to test a string expression. However, the interpreter does not ensure that the expression used in an IF/THEN statement is a number. A string expression can be used in an IF/THEN statement and the interpreter will not detect the error. However, using an IF/THEN statement with a string expression more than two or three times in a program causes a “?FORMULA TOO COMPLEX” error. In addition, the logical value returned for the string is not consistent.

A string expression is not simply an expression that contains string operands. String expressions are defined as expressions that evaluate to a string result. The following examples evaluate to a string result and are string expressions:

```
CHR$(3)
A$+B$
STR$(I*J)
```

The following examples evaluate to a numeric result and are numeric expressions:

```
A$<B$+C$
A$<CHR$(2)
FLAG AND A$<C$
```

The compiler fully supports the use of IF/THEN in all its forms with a numeric argument, but an IF/THEN statement with a string expression is flagged as an error during compilation. IF/THEN is discussed more completely in the *Applesoft II BASIC Programming Reference Manual*.

Numeric GET

The Applesoft interpreter's GET statement was designed for use only with strings. However, it is possible to use it with numeric variables. Unfortunately, the interpreted GET may be inconvenient to use with numeric variables. The biggest problem is that the interpreted numeric GET yields a syntax error and stops the program if the input response is nonnumeric. The compiled numeric GET eliminates this problem. Entering a nonnumeric input does not generate an error message, and it does return a zero as the entered value.

Numeric READ

The Applesoft interpreter does not allow numeric strings to be read into numeric variables. For example, although the following DATA statements are treated identically when they are READ into a string variable, the second generates a "?SYNTAX ERROR" when it is READ into a numeric variable:

```
10 DATA 1234
20 DATA "1234"
```

The compiled version of READ eliminates this inconvenience. However, the compiled version of INPUT still generates a "?REENTER" message when a quoted number is entered. The interpreter's conventions for leading and trailing spaces in literals and strings are implemented without change.

Operational Differences

This section describes several additional differences between compiled and interpreted programs. Most of the differences are not usually significant, but they may need to be taken into account in some cases.

The Runtime Stack

Compiled and interpreted programs use part of memory as a stack to store GOSUB/RETURN and FOR/NEXT information. The routines that are used by a compiled program are more stack-efficient than those in the interpreter. Therefore, some programs that generate stack overflow errors when executed with the interpreter can be run without problems when compiled. Although compiled programs use less stack space than interpreted programs, compiled code *does not* check for stack overflow.

There are 254 bytes of free stack space. RETURN entries take 2 bytes, and FOR entries take 16. The stack overflows if subroutine calls are nested more than 127 levels deep, or if FOR/NEXTs are nested more than 15 levels deep. Part of the stack area is also used as scratch space by the Applesoft PRINT and STR\$ routines. Using these statements overwrites the last 16 bytes of available stack space. Overflowing the stack or overwriting the top of the stack when it contains information causes a compiled program to behave unpredictably. Since compiled programs are more stack-efficient than interpreted programs, observing the restrictions imposed by the interpreter should prevent stack overflow when running a compiled program.

ONERR GOTO and the Stack

The ONERR GOTO statement may cause difficulties with the stack in both compiled and interpreted code. Problems occur when an internal interpreter or runtime routine is exited with an error condition. Most internal routines use a small amount of stack space as temporary storage. Exiting with an error condition may leave some stored parameters on the stack. When ONERR is not in effect, the program stops execution, and the information on the stack does not cause any problems.

When ONERR GOTO is used to handle errors, the extra stack entries are left on the stack and control is transferred to the error-handling routine. The stack is restored to its state before the error and the extra bytes do not cause any problem, if the error-handler ends with RESUME. However, if the error-handler returns without executing a RESUME, the left-over information on the stack is never removed. Repeating this process too many times causes the stack to overflow.

Most problems with the use of the ONERR statement result from inconsistent use of the stack. The *Applesoft II BASIC Programming Reference Manual* includes a machine-language program that restores the stack to its state before the last error occurred. Calling this routine at the end of an error-handler restores the stack and prevents overflow. This routine also resets the stack when used in a compiled program. However, the compiled code does not usually save the stack pointer before each statement. When the routine tries to restore the stack to its previous state, it will set the stack pointer incorrectly.

For this reason, the compiled code must save the stack pointer before each statement. The RESUME statement also requires the compiled code to save information before each statement. Generating the extra RESUME/Debug code is a compilation option, since it slows down execution and makes the program longer.

You do not need to use the RESUME statement at the end of the error-handling section of your program. However, the extra RESUME/Debug code *must* be included in the compiled program if the *Applesoft II BASIC Programming Reference Manual's* stack clear routine is to function correctly. Failure to include the extra code will cause the compiled program to jump into the monitor or to encounter other problems. The stack clear routine should not be used unless absolutely necessary, since including the RESUME code slows the program and makes the object file longer.

Programs that use the stack clear routine can usually be recognized by CALLs in their error-handling routines. See the *Applesoft II BASIC Programming Reference Manual* under ONERR GOTO for information about the stack clear routine. See Chapter 5 of this manual for information on invoking the RESUME option.

Special Machine-Language Calls

A few Applesoft programs use special techniques for passing information to machine-language routines. The most common method is to include extra text following the machine-language call. For instance, the following statements might be used:

```
USR(0)"3,5,6"  
CALL 520 "PROGRAM 2"
```

These statements work with the interpreter only because the machine-language program can change the interpreter's pointer into the current statement and prevent the added characters from being seen. Since the compiler scans all statements at compiletime, the extra text will be flagged as a syntax error. See the section on COMMON in Chapter 8 for more information on passing parameters to machine-language programs.

Using MAXFILES from within a Compiled Program

The DOS MAXFILES command sets the number of available file buffers. The number of available buffers determines how many files can be open simultaneously. Since the compiler fully supports all DOS commands, the MAXFILES command can be used from within a compiled program. However, DOS does not perform the additional operations needed to correctly execute MAXFILES in a compiled program. DOS changes the value stored in the HIMEM location, but DOS does not alter other pointers that must be changed so that a compiled program will conform to the new HIMEM value.

MAXFILES in a compiled program must be followed by a HIMEM statement to set all pointers correctly. The HIMEM statement should simply specify the new HIMEM value provided by the MAXFILES command. This can be accomplished with the following statement:

```
HIMEM: PEEK(115) + 256 * ABS(PEEK(116))
```

Using HIMEM in either a compiled or interpreted program destroys the current string values, so HIMEM and MAXFILES should usually be used as the first statements in a program.

Using RUN with COMMON

The Applesoft RUN command is normally used as an immediate command from the editor, but the interpreter also allows it to be included in a program. RUN in an interpreted program clears all variables and reexecutes the program. RUN with the optional line number specified also clears all variables, but begins execution at the specified line rather than the beginning of the program.

The compiler also includes the RUN command. RUN without the optional line number reexecutes the program by jumping to the ampersand vector. The reexecution causes the same clearing action that began the program. If the program has no COMMON variables, or specified them with USECOMMON, then only local variables are cleared. If the program specified DEFCOMMON, then both COMMON and local variables are cleared.

Use of the RUN <linenumber> form of the RUN command causes a CLEAR, followed by a GOTO to the specified line. COMMON variables are never cleared, since the compiled CLEAR command only initializes local variables. See Chapter 8 for more information on COMMON and CLEAR.

NEW, END, and STOP

The NEW statement in an interpreted program erases the current program before terminating execution. STOP prints the message "BREAK IN ####" before termination, and END simply terminates execution. The compiled NEW and END statements function identically. STOP still prints the BREAK message, but the number specified is an object code address, not a line number. All three commands delete any interpreted program in memory, initialize all Applesoft pointers properly, then reenter the interpreter.

Abnormal Termination of a Compiled Program

Although the compiled and interpreted versions of a program produce the same output, the internal process is substantially different. Both the compiled code and the interpreter make extensive use of memory page zero, which resides at \$00-FF (decimal 0-255). Many of the locations used by the interpreter are used for a different purpose when a compiled program is running. See Appendix E for a description of how the zero page is used by a compiled program.

Compiled programs begin with a call to an initialization routine that sets up the zero page for execution of the compiled program. When the compiled program executes an END, STOP, or NEW statement, or is interrupted by <CTRL-C> during an INPUT statement, it reinitializes page zero for the interpreter before terminating execution. This terminates the compiled program properly.

Using <RESET> to interrupt a compiled program stops the program, but does not reinitialize the zero page for the interpreter. Runtime errors also stop the program without reinitialization. Attempting to use the interpreter at this point is unwise. Even if statements appear to function normally, the interpreter may be destroying DOS or making other errors. A NEW command is necessary to properly reinitialize the interpreter. NEW sets the pointers stored on page zero to their correct values, allowing the interpreter to operate normally.

Applesoft Pointers Preserved by Compiled Code

There are only two pointers used by the interpreter that are preserved during execution of a compiled program. These two pointers are MEMSIZ and TXTTAB. MEMSIZ is the top-of-memory pointer affected by the HIMEM statement, and TXTTAB is the beginning-of-program pointer. They reside at \$73-4 (decimal 115-6) and \$67-8 (decimal 103-4), respectively.

Compiled programs use MEMSIZ for the same purpose the interpreter does. The compiled HIMEM statement changes the contents of MEMSIZ.

Applesoft Compiler

Compiled programs do not use TXTTAB, but the interpreter uses TXTTAB to point to the beginning of the current program in memory. The interpreter and DOS also use TXTTAB to decide where a LOAded program should begin. Compiled programs preserve TXTTAB so that a compiled program can easily RUN an interpreted program using DOS.

The program storage format used by the Applesoft interpreter requires that the location just before the program area pointed to by TXTTAB contains a zero. A program can still be entered when the location is not zero, but attempting to RUN the program produces a "?SYNTAX ERROR." This restriction also holds when using DOS to LOAD or RUN interpreted programs. The location does not usually have to be explicitly set to zero, since TXTTAB normally points to its default location of \$801. \$800 is set to zero when Applesoft is initialized, so leaving TXTTAB at its normal value alleviates any problems.

To maintain this convenience, the compiler leaves location \$800 protected in its default memory allocation mode. Preserving this location allows compiled programs to RUN interpreted programs, without first having to store a zero in \$800. \$801-802 are protected for similar reasons, so the default address for the library is \$803 (decimal 2051).

Linking between Compiled Programs

Interpreted programs are linked using the DOS RUN command. Attempting to RUN an object file produced by the compiler produces an error, since compiled programs are binary files, not Applesoft programs.

Compiled programs are linked using the BRUN command. To facilitate linking, the compiler allows COMMON variables, a powerful extension that is not available in the Applesoft interpreter. Programs executed in sequence can use COMMON variables to pass information. See the next chapter, "Language Enhancements," for an explanation of this new feature.

String Operations

The Microsoft Applesoft Compiler handles strings differently than the interpreter. The interpreter usually duplicates string values for assignments such as `A$=B$`. If thirty strings have the same string value, the interpreter then stores the same string thirty times in memory. Copying strings makes the interpreter especially slow in applications that move strings frequently.

The compiler eliminates string copying by allowing several strings to point to the same value in memory. This eliminates the need to duplicate strings and makes operations like sorting much faster. In exchange for being very fast in assignments, the compiler is slightly slower on operations that build and take apart strings. `LEFT$`, `RIGHT$`, and `MID$` are less efficient. Since these functions are normally used less often than string assignments, the modified method used in the compiler is more efficient.

Garbage collection usually presents the biggest speed problem in string operations. Both the interpreter and the compiler must "houseclean" when the available string space is filled. Garbage collection compacts the strings that are still being used, and eliminates any string "garbage."

The frequency of garbage collection is determined by two factors: the amount of free space available and the rate of garbage production. The amount of free space affects how much garbage can accumulate before collection is necessary. In turn, the rate of garbage production determines how quickly the space is filled.

Garbage collection is normally a lengthy process. The time required depends on how many string variables are used in a program. Large string arrays substantially slow garbage collection, since the program must look through more variables to decide whether a string should be kept or discarded. Each time that garbage collection is necessary, the program must suspend execution to houseclean.

Applesoft Compiler

The number of times that garbage collection is necessary can be reduced by increasing the string space available or decreasing the rate of garbage production. More space can be obtained by reducing the size of arrays, shortening the program, or setting DOS MAXFILES to a smaller number. The rate of garbage production can be decreased by holding operations such as LEFT\$, RIGHT\$, MID\$, and string concatenation (+) to a minimum. The number of assignments is not a factor, because the compiler does not duplicate strings for assignment.

The time spent during each garbage collection call can also be reduced by decreasing the number of string variables. Cutting the number of variables in half usually makes garbage collection about four times faster, because the time required is roughly proportional to the square of the number of string variables. Using string operations efficiently can increase execution speed, and keeping the number of garbage collection calls to a minimum results in the best performance.

Chapter 8

Language Enhancements

The Microsoft Applesoft Compiler provides two major enhancements to Applesoft that substantially increase speed and programming power:

1. True integer arithmetic, including integer FOR/NEXT loops.
2. COMMON variables.

The compiler makes these enhancements possible by including five new statements:

```
CLEAR CHAIN  
CLEAR COMMON  
DEFCOMMON  
INTEGER  
USECOMMON
```

These new statements and the language enhancements they allow are discussed in the sections that follow.

Integer Arithmetic

The compiler includes a full integer arithmetic package. The integer math routines allow very fast operations on integers. They are a major improvement over the methods used by the interpreter.

Integer Arithmetic in Interpreted Programs

The Applesoft interpreter includes the use of integer variables, but it does not actually perform integer operations. All integers are converted to floating-point numbers before being operated on. Since this conversion is necessary each time the variable is accessed, operations on integer variables are actually slower than operations on floating-point variables.

The only advantage to using integers in the interpreter is that elements of integer arrays occupy two bytes of memory rather than five, because Applesoft does not include integer operations. The interpreter also does not allow an integer variable as the index to a FOR loop. These problems combine to make the use of integer variables in interpreted programs of little value.

Integer Arithmetic in Compiled Programs

If real variables are changed to integers, programs compiled with the Microsoft Applesoft Compiler can be made substantially faster. However, converting a program by adding a percent (%) sign to each variable name is time-consuming. To eliminate this problem, the compiler includes the declaration statement `INTEGER`. `INTEGER` allows variables to be defined as integers without the addition of the normal % sign. As an example, the real variable `I` can be declared as an integer by simply including the statement `INTEGER I`.

The `INTEGER` Statement

Including the `INTEGER` statement as a normal statement causes a syntax error if the program is run under the interpreter, since the interpreter does not recognize the `INTEGER` statement. To avoid this problem, the `INTEGER` statement must be included in a special "active" `REM` statement. To allow the compiler to distinguish these special `REM` statements, "active" `REMs` are distinguished from normal "non-active" `REMs` by using an exclamation point after the word `REM`. For example, the following statement declares `I` as an integer variable:

```
10 REM! INTEGER I
```

The other added statements described later must also be disguised in active REMs. The new statements are ignored when the program is run by the interpreter, but they are recognized by the compiler and treated as normal statements.

The INTEGER statement can declare either arrays or simple variables as integers. Arrays are declared by including the name and dimensions in the same form they would be used in a DIM statement. The dimensions specified must be identical to the dimensions in any other INTEGER, DEFCOMMON, USECOMMON, or DIM statements. DEFCOMMON and USECOMMON are discussed later in this chapter.

Multiple variables can be declared as integers by separating the variable names with commas. Spaces are allowed between variable names and commas, but may not be embedded within variable names or array dimensions.

The Microsoft Applesoft Compiler includes a wildcard option for INTEGER, because many programs can declare all their variables as integers. Using an asterisk (*) as the first non-space character following INTEGER causes all numeric variables to be treated as integers. The effect of including INTEGER * is the same as putting the name of every real variable and array in an INTEGER statement.

A variable may be declared as INTEGER several times, but only the first declaration has any effect. Multiple INTEGER statements are allowed. The following declarations are all acceptable:

```
10 REM! INTEGER I,J,K,L
20 REM! INTEGER I, J, AB(3,9), R(3)
30 REM! INTEGER *
```

The following declarations include spaces embedded within variable names, and are therefore unacceptable:

```
10 REM! INTEGER A % ,A 2, A 3
20 REM! INTEGER BA(3, 7), D %(3,7)
```

INTEGER statements can follow or be mixed with normal inactive REMs, but they must precede all other statements in

the program. The compiler scans all REM statements, and ignores those whose messages do not begin with an exclamation mark. It also ignores REMs that do not contain one of the keywords INTEGER, COMMON, CLEAR COMMON, or CLEAR CHAIN. During compilation, the compiler notifies the user that it has processed an active REM by printing "RECOGNIZED" on the screen. This message is displayed even when the listing option is turned off. These messages must be monitored carefully, since an incorrect active REM that is not being RECOGNIZED is otherwise hard to detect.

Integer FOR/NEXT Loops

The interpreter does not allow integer variables denoted by a percent sign (%) to be used as loop variables in FOR/NEXT loops. The INTEGER statement allows the compiler's integer FOR/NEXT loops to maintain compatibility with the interpreter.

Instead of modifying Applesoft syntax and allowing normal integer variables as FOR loop index variables, the Microsoft Applesoft Compiler's integer FOR/NEXTs use integer variables declared with the INTEGER statement. For example:

```
10 REM! INTEGER I
20 FOR I = 1 TO 10 : PRINT I : NEXT I
```

The INTEGER statement solves the problem of interpreter compatibility, since the control variable is treated as a real variable by the interpreter. The compiler recognizes the loop variable as an integer and produces special code for the loop. Since the loop variable is an integer, the initial, final, and step values are considered integers, and must be in the range -32767 to +32767. Integer loops are usually about twice as fast as their real counterparts.

Integer Operations

Integer values in an expression cannot always be operated on in integer mode. If the operation also contains real variables, or if the integer is used as an argument to a function that expects a floating-point value, the integer will be converted to floating-point form.

The use of some integer operations is controlled by the setting of the integer arithmetic option. Addition, subtraction, multiplication, and negation can generate overflows in the integer accumulator, so their use can be explicitly controlled.

If the option is on, true integer operations are performed whenever both operands are integers. If the switch is off, integer operations are only performed for comparisons and logical tests. Since these operations cannot generate an overflow, their use is automatic and is not controlled by the setting of the integer arithmetic option.

Integer operations should be turned off only as a last resort. If including true integer arithmetic causes overflow in only a few cases, the expressions that produce overflows can be forced into real mode. For instance, A can be temporarily assigned to a real variable before the addition and B can be added to the real variable instead, if the integer mode addition of the two integers A and B produces an overflow. Since one of the addends is a real variable, B is converted to floating-point form, and the addition is forced to real mode. This forcing technique requires some small modifications to the program, but the added speed produced by using integer operations elsewhere in the program usually justifies these changes.

Many Applesoft statements and functions use integer parameters. Since the interpreter treats all values as floating-point numbers, it must compute the parameters as floating-point numbers, then convert them to integers. Similarly, a few functions return integer results and the interpreter must convert the results back to floating-point form. These continual conversions often slow interpreted programs.

Functions and statements that expect or return integer values can be made much faster by using them with integer expressions, since compiled code can evaluate expressions

Applesoft Compiler

in integer mode. Graphics statements, string functions, and game controls all expect integer parameters. Using integer expressions as the arguments to these operations results in a significant increase in speed. Eliminating the conversions that are normally necessary often doubles the speed of the operation. Games and graphics programs usually benefit substantially from this technique. The compiled code always converts parameters if necessary, but the conversion makes the operation slower.

The following operations expect integer values:

CHR\$	PLOT
COLOR=	POKE (2)
DRAW	PR#
FOR (integer)	RIGHT\$
HCOLOR=	ROT=
HLIN	SCALE=
HPLOT	SCRN
HTAB	SPC
IN#	SPEED=
LEFT\$	subscripts
LET (integer)	TAB
MID\$	VLIN
ON GOSUB/TO	VTAB
PDL	WAIT (2 & 3)
	DRAW

Numbers in parentheses indicate which parameters are treated as integers when the operation has parameters of mixed types.

The integer arithmetic package always performs integer comparisons when both operands are integers. Since division usually produces fractional results, division is always performed in real mode. The following operations can be performed in integer mode when all operands are integers and the integer math package is turned on:

- addition
- negation
- multiplication
- subtraction

The following operations can take either floating-point or integer values without forcing conversion:

AND	NOT
FRE	OR
IF/THEN	POS

All other operations expect floating-point values.

Variables should be typed as floating-point or integer, depending on which way they are used most often in the program. Failing to convert variables that can normally be treated as integers will unnecessarily slow the compiled code. On the other hand, if a variable is normally needed in floating-point form, it should be left as a real variable. Matching the types of the parameters supplied to the types expected will result in appreciable speed increases. The following operations return integer values:

ASC
LEN
PEEK
PDL
POS
SCRN

The values returned by these operations should be treated as integers whenever possible to increase speed and prevent conversions.

Using the integer features provided in the compiler will significantly improve the execution speed of compiled programs. *The importance of effectively using integers cannot be overemphasized.*

Chain with COMMON

The Microsoft Applesoft Compiler provides the ability to pass COMMON variables between chained programs, which is another important extension to Applesoft. The addition of COMMON allows several programs to pass values without having to alternately store and recall the values to and from disk. The implementation of COMMON in the compiler is normally referred to as "blank" COMMON. Blank COMMON uses a single COMMON block. COMMON is implemented by allocating a static block of memory for the COMMON variables to occupy. This block of memory is left protected as the chained programs are BRUN in succession. Each program refers to the same block of values as it executes. Variables that are declared as COMMON in a program are allocated in the block in a specific order, so that variables in different programs can correspond.

The actual names used to refer to the variables do not matter. The common memory allocation is responsible for the correspondence. For instance, assume PROGRAM1 declares variable A1 as its first COMMON variable, and PROGRAM2 declares variable A2 as its first COMMON variable. Since the location of the two variables in the COMMON block is the same, the value left in variable A1 at the end of PROGRAM1 is the initial value of variable A2 at the beginning of PROGRAM2.

USECOMMON and DEFCOMMON

The DEFCOMMON and USECOMMON statements are used to declare COMMON variables. Both forms of the COMMON statement must be included in an active REM as follows:

```
10 REM! DEFCOMMON I,J,K
```

Multiple COMMON statements are allowed. Any INTEGER declarations must precede COMMON statements, and both must precede any other statements (except normal inactive REMs). Incorrect ordering produces the "DECLARATION" error during compilation. COMMON may specify any type of variable in either simple or array form.

Variables may be declared in both INTEGER and COMMON declarations. Specifying a variable more than once in a COMMON statement generates the "DECLARATION" error during compilation. Arrays are specified in the same way as they are with INTEGER, and the same rules hold; any DIM statements must specify the same dimensions for the array.

DEFCOMMON sets up a COMMON block and initializes the variables in it. USECOMMON accepts a COMMON block already set up by another program without clearing the block. DEFCOMMON presents a problem when several programs chain back to a main menu. Using DEFCOMMON in the menu would cause information returned from the subprograms to be erased, but USECOMMON would fail to initialize the COMMON block at all.

The solution is to add another program containing DEFCOMMON that chains to the menu program, and then use USECOMMON in the menu, as Figure 2 illustrates:

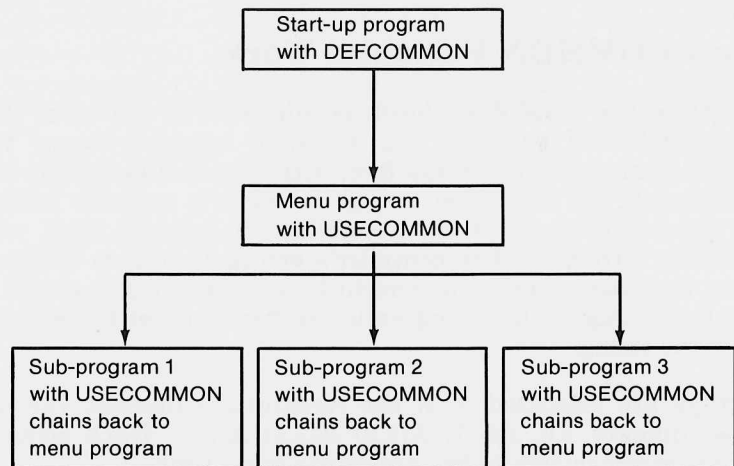


Figure 2. System of Programs Using DEFCOMMON and USECOMMON

The start-up program is initially executed to set up the COMMON block. It then runs the menu program and is *never returned to*. This arrangement prevents the DEFCOMMON from erasing the COMMON variables each time you chain back to the menu.

CLEAR CHAIN and CLEAR COMMON

The compiled version of the Applesoft CLEAR statement does not clear COMMON variables, since COMMON variables are normally intended to be passed to a later program. The compiler provides an additional statement, CLEAR COMMON, that clears COMMON variables. CLEAR COMMON does not affect local variables. CLEAR COMMON must be used in an active REM.

Programs chained with COMMON are linked as usual by using the BRUN command. However, an additional CLEAR CHAIN statement must be included immediately before the BRUN statement. The CLEAR CHAIN command sets up string storage so that strings are properly preserved across the chain. Failure to execute a CLEAR CHAIN causes some string values to be lost or modified. CLEAR CHAIN forces garbage collection and reinitializes all local variables. CLEAR CHAIN must be included in an active REM on the line before the BRUN command.

How COMMON Variables Work

Space in the COMMON block is allocated in the order that the COMMON variables are declared. Integers occupy two bytes. Real numbers occupy five. String variables require two bytes each; but since they are allocated in a separate portion of the block, there is no possibility of mixing them with numeric variables. The compiler's string format is different from the interpreter's. The length byte for a string is stored at the beginning of the string value, rather than with the pointer to the string.

Arrays are allocated with the rightmost subscript varying most quickly, i.e, A(1,1) A(1,2) A(2,1) A(2,2). Each element occupies two or five bytes, according to its type.

Space in the COMMON block is divided into two subblocks: numeric variables and strings. Within each subblock, the order of variable space allocation is determined by the order in which the variables are declared in COMMON statements. When the compiled code is executed, it checks to make sure that the size of the numeric and string COMMON blocks it expects is identical to the size of the blocks actually passed to it by the previous program.

This is the extent of the type-checking performed. The compiler does not protect or prevent the user from accessing the ten bytes declared as two five-byte real numeric variables in one program as five two-byte integer variables in another program. This type of error can be easily avoided by making the COMMON declarations in the two programs identical. Mismatched COMMON block sizes produce the “?TYPE MISMATCH” error at runtime.

COMMON can also be used to pass parameters to machine-language routines. The beginning of the block and the positions of the variables within the block are fixed, so machine-language programs can reference static locations for the variables. The compiled program can also POKE the variable values into a predetermined location before calling the machine-language routine. Some interpreter routines locate variables by looking them up in the interpreter's variable list. These routines will not work with a compiled program, since the variable list is eliminated. These programs must be modified to use COMMON or POKE values explicitly.

Notes on COMMON

Space for the COMMON block is allocated at the beginning of the space declared for program storage. Programs that share a COMMON block must all be compiled with the same starting address for the program space. As the compiler scans the COMMON statements, it increments the starting memory address of the compiled program to leave room below it for the COMMON variables. Including a GOTO or transfer to one of the REMs that declare the COMMON block causes the compiled program to jump into the COMMON block, producing undefined results.

Using the interpreter to debug programs that use the compiled COMMON statement is difficult, since COMMON is not included in the Applesoft language. However, DOS provides a machine-language program that can simulate most of the facilities of the compiled chain with COMMON. See the DOS Version 3.3 manual under CHAIN for more information.

An alternative debugging method is to simulate the COMMON block by using disk files. PRINTing and INPUTing the COMMON variables in the same order that they occur in the COMMON block will help to detect any ordering errors. This method requires modifying the programs, but it also provides a reasonably good simulation of the compiled COMMON.

Creating a System of Programs

The DEFCOMMON and USECOMMON statements are designed for creating large systems of Applesoft programs that communicate with each other. A sample situation is described here to show possible interactions in a large system of programs. The distinction between BRUN with COMMON and a simple BRUN is also demonstrated.

Consider the following integrated accounting system containing three packages for general ledger (GL), accounts payable (AP), and accounts receivable (AR). Entry into each package is controlled by a main menu program. The system structure is:

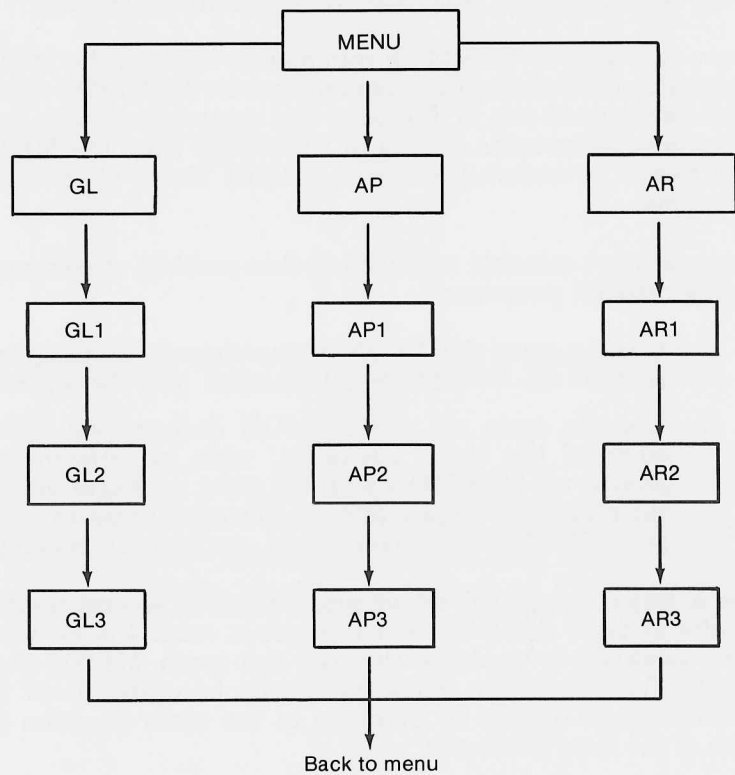


Figure 3.
System of Programs Controlled by a Main Menu

In order to use COMMON effectively, it is important to structure the system and the COMMON information logically. In the system shown in Figure 3, the subprograms within each of the three packages must pass information to each other.

There may also be COMMON information between the MENU and each of the packages. However, because COMMON blocks must be the same size in programs that communicate with each other, any information passed in COMMON from the MENU also has to be included in the COMMON blocks of the other programs.

There are two possible solutions to this problem of communicating between programs:

1. Use the same COMMON declarations in all programs so that all COMMON information may be shared.
2. Use the same set of COMMON declarations within each of the three packages, with no information shared via COMMON with the other packages or with the main MENU program. In this case, there are three sets of COMMON declarations, one for each package.

For a large, integrated set of programs, the second method results in more flexibility, since program control is switched from package to package through the main MENU. Any input information that would ordinarily be passed from the MENU would instead be obtained in the main program for each of the three packages.

The following program fragments demonstrate how the programs in the figure might be linked. The short fragments also provide examples of when to use a simple BRUN and when to use BRUN with COMMON.

MENU

```

:
1000 INPUT "WHICH PACKAGE?" ; N
1010 IF N = 1 THEN PRINT D$ "BRUN GL"
1020 IF N = 2 THEN PRINT D$ "BRUN AP"
1030 IF N = 3 THEN PRINT D$ "BRUN AR"

      GL

10  REM! DEFCOMMON A, B(3,4), C$, D$
:
1000 REM! CLEAR CHAIN
1010 PRINT D$ "BRUN GL1"

      GL1

10  REM! USECOMMON A1, B1(3,4), C1$, D$

      GL1

10  REM! DEFCOMMON A, B(3,4), C$, D$
:
1000 REM! CLEAR CHAIN
1010 PRINT D$ "BRUN GL2"

      GL2

10  REM! USECOMMON A2, B2(3,4), C2$, D$
:
1000 REM! CLEAR CHAIN
1010 PRINT D$ "BRUN GL3"

      GL3

10  REM! USECOMMON A3, B3(3,4), C3$, D$
:
1000 PRINT D$ "BRUN MENU"

```

The examples are shown for the GL package. The other two packages would be similar. Notice that the MENU does not have a COMMON declaration, since it does not pass or receive information. GL has a DEFCOMMON declaration because it must pass information on to GL1, but it does not receive information from the MENU. Declaring DEFCOMMON sets up the COMMON block and initializes it.

Applesoft Compiler

GL uses a CLEAR CHAIN before the BRUN, since GL must pass information on to GL1. GL1 declares USECOMMON, since GL1 must receive and not initialize the COMMON block passed by GL.

GL1 includes a CLEAR CHAIN in order to pass information on to GL2. GL2 is similar to GL1, and passes the COMMON information on to GL3. GL3 also declares USECOMMON so that it can accept the values passed to it by GL2. GL3 does not need a CLEAR CHAIN before the BRUN, since no information is passed from GL3 back to the menu.

Chapter 9

How the Compiler Works

This chapter explains how the Microsoft Applesoft Compiler works, and should make the internal operation of the compiler less mysterious. The primary function of the compiler is to translate the source program into machine-language. This process is divided into two basic steps:

1. Syntax Analysis

The recognition of Applesoft statements that the compiler takes as input.

2. Code Generation

The production of machine-language equivalents for the Applesoft statements.

Discussion of these two major steps is preceded by a description of the programs that perform them.

PASS0, PASS1, and PASS2

The Microsoft Applesoft Compiler is a “two-pass” compiler, since it compiles in two major steps. PASS0 simply picks up user input and sets up compilation parameters, so it is not really part of the actual compilation process. The Applesoft program APCOM runs PASS0.

PASS0 and PASS1 chain to PASS1 and PASS2, respectively. All three passes were written largely in Applesoft, and APCOM was used to compile itself.

PASS1 is the first pass. PASS1 performs syntax analysis and generates most of the code. As it examines the program, PASS1 also collects information about variables and line numbers and stores the information in a symbol table.

The symbol table is used to store all the information about the program. "Compiling" all this information and inserting it where it is needed eliminates the runtime searching that occurs during interpretation.

PASS1 cannot allocate variable storage space or know the addresses of all the line numbers until it has processed the entire program. Therefore, it cannot insert actual addresses into the code it generates. Instead, it must leave a trail so the addresses can be patched in later. PASS2 performs this patching after PASS1 has finished processing the source file.

PASS2 uses the information provided by PASS1 to allocate variable storage space. PASS1 keeps a record of variable usage in the symbol table. PASS2 uses the symbol table to allocate storage space. PASS2 uses the stored variable types to decide how much storage space to allocate. PASS2 then saves the address of the storage space allocated for each variable with the other information about the variable.

Line numbers are handled differently. Keeping all line numbers and addresses in memory requires too much memory, so PASS1 stores the addresses of the line numbers in the disk file "CLINENUM" as it generates the code. PASS2 uses this file to match line number references to their actual addresses.

When the actual addresses are determined, PASS2 must find all the references left undefined in PASS1. Rather than keeping a huge list of all the locations in the program where each variable is used, PASS1 links all the references for each variable or line number into a chain. Each reference contains the address of the previous reference, rather than the unknown address.

Because the file is processed forward, these chains lead backward from the current position toward the beginning of the file. PASS1 keeps the most recent reference in the symbol table. The symbol table reference points backward to the next most recent, that one points to the one before it, and so on.

This linking leaves PASS2 with the end of the chain for each set of references, and PASS2 uses the chains to trace backwards and patch the undefined references. This process is described as "backpatching," and it is the sole purpose of PASS2. Backpatching disk files is necessarily a slow process, but building the files on disk eliminates the program length limitations imposed by RAM-based compilation.

Syntax Analysis

Syntax analysis is the first step in the compilation process, since the compiler must be able to understand the source program before it can generate code. The syntax analyzer must examine the current line of input and decide what statement it represents before it can tell the code generator what kind of code to produce.

The process of recognizing statements can be broken into two parts: lexical analysis and parsing. These two steps are demonstrated by reading a sentence. Grouping letters into words is lexical analysis; grouping words into sentences is parsing.

Lexical Analysis

The lexical analyzer looks at the characters in the input line and separates out the words that are part of Applesoft statements. These words, like PRINT, FOR, etc., are called "key-words." A lexical analyzer might take "FOR I = AB TO C" and produce "FOR I = AB TO C". The lexical analyzer substitutes a numeric "token" for each keyword it finds, making the keywords easy to recognize later. Tokenizing "FOR", "=", and "TO" in the previous example might produce "<129> I <208> AB <193> C".

The interpreter lexically analyzes each program line as it is typed in, so Applesoft programs are stored in "tokenized" format. When a program is LISTed, the interpreter substitutes the appropriate word for each token, and the program comes out in readable form. Programs stored on disk are also in tokenized format.

Parsing

Since the Applesoft interpreter tokenizes programs, the compiler does not have to perform lexical analysis. However, the compiler must still parse the input. Parsing is normally accomplished by one of two approaches: top-down or bottom-up. Both methods produce the same result: they take input that has been lexically analyzed and then identify logical groups of information. Top-down parsing first assumes that the in-

Applesoft Compiler

put is a certain statement, then attempts to find the parts that should be there. Bottom-up parsing first examines the parts, then deduces the type of statement they represent. Most Applesoft statements can be identified by looking at the first character, so the compiler parses statements from the top down.

Expressions must also be parsed. Although they are not composed of letters and words, expressions still include symbols that the compiler must be able to organize and understand. The parser uses operator precedence rules and any parentheses present to decide what operations must be performed first.

The code generator does not consider precedence, so the parser has to reorder the operations so they can be output sequentially during the code generation. Expressions are parsed bottom-up, and reordering is accomplished with a stack.

The reordered expressions are passed to the code generator in the form of "triples." A triple is a unit consisting of an operator and one or two operands. Operations or functions that have only one operand leave the space for the other unused. The operand of one triple is often the result of a previous triple. The following set of triples represents the expression $A + B * C$:

Triple #	Operator	Left Operand	Right Operand
1	load	B	
2	load	C	
3	multiply	triple 1	triple 2
4	load	A	
5	add	triple 4	triple 3

Triples #1 and #2 simply indicate that the variables B and C must be loaded. Triple #3 instructs the code generator to generate a multiplication of B and C. Triple #4 indicates that the value for A must be loaded. Triple #5 specifies that A should be added to the result produced by the multiplication in triple #3.

Code Generation

Code generation takes place in two different ways. Code for expressions is generated by an explicit call to the code generator with a set of triples. Code for statements is generated during syntax analysis. For instance, the call to the CLEAR routine is generated as soon as the syntax analyzer recognizes the CLEAR statement. This method is called syntax-directed code generation. Most statements like END, GR, TEXT, etc., are represented by a single call to a library or Applesoft routine.

Statements that involve expressions are normally single calls also. The syntax analyzer recognizes the statement, then parses the expression. A call to the code generator produces the instructions to evaluate the expression. The syntax analyzer then finishes the code for the statement with a machine-language call to the routine that uses the value.

The interpreter must search lists of variables and line numbers to find a variable or line number. The compiler, on the other hand, generates absolute addresses for variables and line numbers. The compiler is faster than the interpreter because it uses these absolute addresses instead of searching long lists.

The compiler collects all the information about the program at compiletime, rather than searching at runtime. Instead of searching for a line number when it encounters a GOTO 80 statement, the compiled code simply jumps to the address already provided by the compiler. Compilation allows program execution without searching through lists. However, the compiler must perform all searching and organization at compiletime.

Special Techniques

The Microsoft Applesoft Compiler was designed with minimal object code expansion as one of the main objectives. Execution speed of the object code was also considered, but a slight

decrease in speed is occasionally compensated for by a worthwhile reduction in object code size. The compiler is designed for all types of programs, but it is intended to be especially beneficial for large programs and systems. Most compilers produce object code that is significantly larger than the original source. This may create difficulties when the program to be compiled is already large. Speed is important, but a fast program that won't fit in the available memory is worthless. Therefore, the compiler uses several special techniques to generate compact code.

Variable Accessing

Variable accessing is a typical cause of code expansion. Compiled and interpreted code must both load and store values to variables. Variable accessing is required almost constantly, so most compilers use general purpose load and store subroutines. Using subroutines saves quite a bit of space. The routine used to transfer a value to and from memory need be included in the object code only once, rather than being duplicated each time it is needed.

Since these subroutines handle the load and store operations for all the variables in the program, they must be given information about which particular variable to load or store. The information passed to the subroutine is normally the memory address of the variable. The address of the variable to be operated on must be passed to the subroutine each time the routine is called. The address is normally passed in two of the microprocessor registers. Every time a variable is accessed, the same process must be repeated: load the registers, call the routine, load the registers, call the routine.

Setting up the registers to pass the address more than doubles the amount of code required for each variable access. There is a more space-efficient way to perform the same function. Instead of passing an address to a general purpose routine, it is more effective to have a special routine *for each variable* that performs the load and call.

The special subroutine for the variable takes the same amount of space as one load and call to the general purpose routine, but now each access for that variable can simply call the specialized subroutine. The subroutine does not need to be passed an address to indicate which variable it should operate on, since it is dedicated to only one variable. With the new specialized routines for each variable, the address no longer needs to be loaded before each call.

This technique takes more space if the variable is only referenced once, but saves space if the variable is referenced more than once. For example, referencing a variable twenty times takes only 67 bytes instead of the normal 140. Unfortunately, this technique works well only when the variable is referenced in one way. Since a variable can be loaded, stored, added, etc., there really should be a specialized routine for each variable *for each operation*. This would substantially reduce the space savings, since several routines would be required for each variable.

The Microsoft Applesoft Compiler uses a more sophisticated method to overcome this problem. Rather than generating several routines for each variable, the compiler generates only one routine for each variable. This single routine has a different entry point for each type of operation on the variable.

Each of the different entry points loads a different number to indicate what operation is needed. All then use the same code to load the variable address. However, because the subroutine is now used to perform several different operations, it can no longer jump off to a single operation at the end. Instead, the subroutine jumps to a special library routine, and the library routine uses the operation number to dispatch and perform the correct operation.

These specialized routines are a very effective way of solving the code expansion problem. The idea of using specialized routines for each variable was introduced by the authors of the Microsoft Applesoft Compiler, which is the first compiler to use this technique.

The RUNTIME Library

The compiler uses other techniques to reduce the size of the object code. Most operations have been moved out of the object code and into subroutines in the library. This increases the size of the library, but reduces the size of the object code. This also allows extensive use of subroutine calls in the compiled code, and produces unbeatably compact object code with only a slight reduction in speed.

The RUNTIME library included with the compiler has a second advantage. It reduces the amount of disk storage space required. Rather than keeping needed routines with each program, a general purpose library is kept as a separate file. This allows all programs to use a common library, and saves each object code file from having to include some of the same routines. With many programs on one disk, this produces a substantial space savings.

One disadvantage is that the whole library is used for every program. For programs that don't use any strings, for instance, the string routines are still included. However, large programs typically use most of the routines in the library, and small programs have enough memory left over so that including the extra routines does not present a problem. The library also fits well below the first HIRES screen, providing a way to effectively use space that might otherwise be wasted.

Chapter 10

Error Messages and Debugging

This chapter explains the error messages that can occur either at compiletime or runtime.

Compiletime Error Messages

The Microsoft Applesoft Compiler produces two types of error messages; warnings and fatal error messages. Fatal errors indicate problems that prevent successful compilation. Warnings simply indicate the use of statements that are ignored by the compiler, such as unsupported Applesoft features or unexecutable code.

Warnings that indicate unexecutable code are usually caused by statements following a GOTO, or RETURN on the same program line. Warnings do not prevent an object file from being created, but no code is generated for the flagged statements. Unsupported statements are simply ignored, and no code is generated for them. RESUME is ignored, unless the RESUME compilation option is enabled.

Fatal errors yield both a message and an error token. The error pointer, !ERR!, appears in the incorrect statement at the point where the error was recognized. Fatal errors cause the incomplete object file to be deleted. Compilation continues only so any other errors can be detected; code generation does not continue. The following list describes the fatal errors and their causes.

Applesoft Compiler

DECLARATION	<p>INTEGER or COMMON declarations out of sequence or not at beginning of program.</p> <p>USECOMMON and DEFCOMMON both declared in a single program.</p> <p>Variable declared as COMMON more than once.</p>
INCOMPLETE	<p>Incomplete expression.</p> <p>Missing right parenthesis in expression.</p>
OPERAND	<p>Illegal operand in expression.</p> <p>Arithmetic constant too large.</p>
REDEFINED	<p>Function defined more than once.</p> <p>Specified array dimensions different from the first dimensions specified.</p>
SUBSCRIPT	<p>First subscript missing.</p> <p>Dimension not an integer constant.</p> <p>Dimension negative or greater than 32767.</p> <p>More than 88 subscripts.</p> <p>Different number of subscripts than in first usage.</p>
SYMBOL TABLE FULL	<p>Compiler out of symbol table space.</p> <p>See page 24, "Compiling Large Programs."</p>
SYNTAX	<p>Missing or added character or item.</p> <p>Line number greater than 65534.</p>
TOO COMPLEX	<p>Expression too complex.</p> <p>IF/THEN nesting too deep.</p>

TOO LONG

Input line longer than 240 characters.

Object code or variables for compiled program extend past 48K. See page 24, "Compiling Large Programs."

TYPE
MISMATCH

Numeric expression where string was expected, or vice versa.

String expression in IF/THEN.

Undefined line numbers or functions produce fatal errors at the beginning of PASS2. The line number given in the error message is the *last* reference to the function or line. Any other references must be corrected as well.

The compiler expects a source file that has been tokenized and has had spaces removed by the interpreter; the compiler does not allow extra spaces. Additional spaces will cause syntax errors during compilation. If a file does have added spaces in it, the spaces can be removed by LISTing the program into a text file, then EXECing the text file back into memory. See the DOS Version 3.3 manual under EXEC for more information.

Runtime Error Messages

The error messages produced by an executing compiled program are identical to those provided by the interpreter, with the exception of the "?TYPE MISMATCH" error. The "?TYPE MISMATCH" error in its normal sense never occurs at runtime, since type checking is performed during compilation. The "?TYPE MISMATCH" error is used instead to indicate mismatched COMMON blocks. See page 52, "CHAIN with COMMON" for more information.

Since the compiled code is a machine-language program, it does not print line numbers in error messages. However, whenever it can, it provides an object code address. This object code address can be matched with the corresponding line in the Applesoft source program by using the line-number-

Applesoft Compiler

to-object-code-address table offered at the end of PASS2. If a program has errors, it should be debugged again with the interpreter.

Some errors that occur at runtime cannot include an object code address. These errors are printed with an address of 0. Using the ONERR/Debug option makes it possible for the compiler to always include an object code address. However, including this option produces a longer and slower object file. For this reason, program development and debugging should be done with the interpreter whenever possible.

Runtime errors cause a compiled program to halt without correctly reinitializing the interpreter. Typing NEW after a runtime error ensures that the pointers and locations used by the interpreter are set up correctly.

Sources of Common Problems

The compiler is exceptionally compatible with the Applesoft interpreter, but there are still some problems that may occur with compiled programs. Chapter 7 explains most of the language differences between compiled and interpreted programs. This section is a troubleshooting guide that discusses other problems that might arise.

The Applesoft Interpreter

Compiled programs can only be executed when the Applesoft interpreter is available at its normal location in memory. Attempting to execute a compiled program without the Applesoft interpreter prevents the compiled code from running. Attempting to execute a compiled program from Integer BASIC rather than Applesoft produces the same problem. The Applesoft interpreter may be present in any form: ROM, RAMCard, Apple Language Card System, etc. Compiled code will *not* run with the old version of Applesoft that loads in as an Integer program.

HIRES Graphics

The compiler fully supports all HIRES graphics without modification, but the HIRES screens and shape tables often cause memory conflicts. Applesoft uses two areas of memory that correspond to the two HIRES "screens," HGR and HGR2. HGR resides in the area of memory from 8192 to 16383 (hex \$2000-\$3FFF), and the area from 16384 to 24575 (hex \$4000-\$5FFF) is used by HGR2. These areas of memory are mapped to the screen to display the HIRES graphics.

The HGR and HGR2 commands write zeroes through the appropriate area of memory. They also destroy any part of a compiled program, library, or variables that extend into this area. If part of one of these blocks is destroyed, the compiled code behaves unpredictably. This problem is usually easy to recognize and correct. The problem occurs only with programs that use HIRES graphics, and usually occurs shortly after these programs attempt an HGR or HGR2 statement.

Problems with the HIRES screens can be prevented by examining the statistics provided at the end of compilation. The addresses included indicate where the program, library, and variables reside. If the program uses HIRES graphics, check the addresses to insure that they do not conflict with the addresses given above for the appropriate HIRES pages. If there is a conflict the program must be recompiled, using the alternate memory allocation methods discussed in Chapter 5, "Compilation."

Shape tables used by HIRES graphics are often the cause of more subtle problems. Shape tables must be BLOAded or POKEd into memory just like machine-language programs, and often present the same memory conflict problems. The shape table may reside in the same place that the compiled code is expected to occupy.

In fact, shape tables for interpreted programs tend to be stored exactly where programs compiled to fit around the HIRES pages reside. The interpreted program normally fits below the HIRES page, so the shape table is often kept just above the page. However, this is exactly where the program resides if it is compiled using the HGR1 and HGR2 memory allocation options. When shape tables are used, their location

can usually be determined by looking at the addresses specified in the POKES that put them in memory. If there is a conflict, it is usually simpler to relocate the shape table. However, the program can also be compiled at a different address by using the information in Chapter 5.

Normal LORES graphics do not present a problem. The memory mapped to the screen by LORES graphics is the same as the memory mapped in normal text mode. This area is out of the way of both compiled and interpreted programs.

Machine-Language Programs

Machine-language programs used with Applesoft often depend on variables or program lines residing in a specific place. Since compilation changes the internal representation of the program totally, some machine-language programs simply do not work with compiled code.

When a machine-language program does not work because it depends on characteristics of the interpreted program, the machine-language routine must usually be rewritten. However, some machine-language programs create problems only because they present memory allocation conflicts.

Machine-language programs are usually written and assembled to reside at a certain address. The Applesoft program normally puts the machine-language program in memory by BLOADing or POKEing it. CALLs, POKES, PEEKs, and BLOADS often indicate the use of a machine-language program.

Some machine-language programs are also tacked onto the end of Applesoft programs, so the machine-language routine is loaded in with the Applesoft file. These machine-language programs are harder to detect, and make successful compilation difficult. "PENNY ARCADE" on the Apple demonstration disk is an example of this type of program. The original author is usually the only one who can easily straighten out these programs.

Machine-language programs are normally CALLED from the Applesoft program. The addresses specified in the CALLs or POKEs are usually a good indication of whether or not the machine-language programs will cause problems. Most routines are written to reside in page three, which is unused by both interpreted and compiled programs. Parts of page three are used by DOS, but the majority of it is free for use. Page three occupies locations 768-1023 (hex \$300-3FF).

Routines located in page three usually do not cause problems. Since page three is not used by the compiled code, the machine-language program does not present a memory conflict. However, some routines still present compatibility problems. CALLs in ONERR GOTO routines usually indicate a special Applesoft routine that is used to clear up problems with ONERR GOTO. See Chapter 7 for more information about the ONERR machine-language routine.

Self-Modifying Programs

Some interpreted programs achieve special effects by modifying themselves as they run. "PHONE LIST" on the Apple demonstration disk is an example of this unusual technique. The interpreted version of "PHONE LIST" POKEs the entered names and addresses into DATA statements inside itself, saving the data by modifying itself as it runs. Other programs "hide" parts of themselves to speed execution. These programs do not work properly when compiled. The only way to compile these programs is to rewrite them more straightforwardly.

Appendix A

Moving Binary Files with the ADR Utility

Binary Files

The Microsoft Applesoft Compiler compiles Applesoft files to produce machine-language files. Since the machine-language files are not Applesoft programs, their format on disk is different from that of an Applesoft program. This section describes how to load and save the machine-language files.

DOS currently includes three file types: Applesoft, Text, and Binary. The type of a file is indicated by a single letter in the CATALOG. File types are indicated by an "A", "T", or "B", which represent Applesoft, Text, and Binary files. The input files to the compiler are Applesoft programs, so they are indicated by an "A" in the catalog. The machine-language files produced are stored in binary format, and they are indicated by a "B" in the catalog.

Saving and Loading Binary Files

The usual LOAD <filename>, SAVE <filename>, and RUN <filename> DOS commands work with Applesoft files, but they cannot be used with binary files. DOS provides a corresponding set of commands for binary files—BLOAD, BSAVE, and BRUN. The "B" prefix to the commands denotes a binary file. The commands function much the same as the corresponding commands for Applesoft files, but there are some differences.

Applesoft Compiler

DOS SAVES Applesoft programs by using information from the Applesoft interpreter. The interpreter provides the information about the beginning and length of the program to be saved, and DOS writes the program to disk. DOS saves the length information with the file so the program can be LOADED back into memory later. The user does not have to worry about telling DOS where the program starts or how long the program is: the Applesoft interpreter provides DOS with all the necessary information.

The Applesoft interpreter cannot provide the information necessary to load and save machine-language programs. The interpreter keeps track of where the current Applesoft program resides, but it does not retain information about machine-language programs. Therefore, the user must specify additional information to BLOAD and BSAVE machine-language programs.

The BSAVE command is similar to the SAVE command for Applesoft programs. BSAVE saves a portion of memory to disk in binary format. Compiled programs are treated like any other binary data in memory. Since the Applesoft interpreter can no longer provide the beginning and length information necessary, the user must specify the starting address and length of the memory to be BSAVED. The normal syntax for the BSAVE command is:

```
BSAVE <filename>, A <address>, L <length>
```

The two letters "A" and "L" precede the numbers that indicate the starting address and length of the memory to be saved. The numbers can be specified in either hexadecimal or decimal format. Hexadecimal numbers are base sixteen, and decimal numbers are the normal base ten. If hexadecimal numbers are used, they must be preceded by a dollar sign "\$". Decimal numbers are used without any special characters. The usual slot, drive, and volume parameters can also be used with BSAVE, BLOAD and BRUN.

BLOAD and BRUN are similar to the Applesoft file commands LOAD and RUN. LOAD for Applesoft programs simply moves the program into memory; RUN LOADs the program, then RUNs it. Similarly, BLOAD loads a section of memory from a binary disk file; BRUN BLOADs the section,

then CALLs the beginning of the section to execute it. Since the binary files stored by the compiler are machine-language programs, BLOAD will load a compiled program; BRUN BLOADs the program and runs it. As explained in Chapter 6, RUNTIME must be in memory before a compiled program can be BRUN.

BLOAD normally loads the contents of the disk file back into the same area of memory from which the file was BSAVED. For instance, BLOADing a machine-language program that was BSAVED starting from location 4000 loads the program back in at location 4000. However, it is also possible to BLOAD a disk file to a different location in memory. The "A" parameter can be used to specify a different starting address for the file. Typing "BLOAD <filename>, A 6000" always loads the binary file into memory starting at location 6000.

The "A" parameter cannot be used to load and run an object file at an address different from the one at which it was compiled. Compiled programs will not run correctly if they are run at this different address.

The "A" parameter with BLOAD is normally used only to BLOAD the RUNTIME library at an address different from its default. The default address for the library is 2051 (hex \$803), and because the library was BSAVED to disk from that address, it normally BLOADs at that address. Specifying the "A" parameter allows the library to be loaded at an address different from its default. The "A" parameter is also accepted in the BRUN command, but as just mentioned, compiled programs cannot be BRUN at an address different from their normal location. Neither BLOAD nor BRUN can include the "L" parameter, since the whole disk file is always loaded. See the DOS Version 3.3 manual under "Binary Files" for more information about binary file commands.

The ADR Utility

Moving compiled programs from disk to disk is not as simple as moving Applesoft programs, since the address and length of the memory range must be specified in the BSAVE command. The compiled program can still be BLOADED without specifying any extra information, but it cannot be BSAVED without knowing its beginning address and length. The ADR utility is included on the Microsoft Applesoft Compiler disk to make finding the required information simple.

DOS stores the address and length of the most recently BLOADED or BRUN file in two special locations. The ADR utility looks at the contents of these locations and prints the beginning and length of the most recently loaded file. ADR is a text file, so it should be executed by typing "EXEC ADR." ADR does not affect any program or file in memory; it simply EXECutes a PRINT statement to provide the needed information. Since the PRINT statement is only understood by the Applesoft interpreter, ADR must be EXECed from the Applesoft interpreter. ADR will not work from the monitor.

ADR prints out the decimal beginning address and length of the most recently loaded file. The numbers printed should be used with the "A" and "L" parameters to BSAVE the machine-language program to disk. The normal sequence for moving a program from one disk to another is:

```
BLOAD <filename>
EXEC ADR
BSAVE <filename>,A<address>,L<length>
```

The address and length for the BSAVE are found by EXECing ADR and are simply included in the BSAVE, preceded by "A" and "L." The same procedure can be used to move the other binary files on the compiler disk—RUNTIME, PASS0, PASS1, and PASS2.

The program CREATE ADR is included so that the ADR file can be used on other disks. ADR is a text file and cannot be LOADED and SAVED or BLOADED and BSAVED to be transferred to another disk. Instead, the CREATE ADR program must be used to write a copy of the text file onto the new disk.

The procedure for transferring the ADR file to a new disk is simple:

1. Load the CREATE ADR program from the Microsoft Applesoft Compiler disk.
2. Remove the compiler disk.
3. Insert the disk that the new copy should be created on.
4. Type RUN.

The CREATE ADR program opens up a text file called "ADR" on the disk, writes the PRINT commands into it, then closes the file and stops. The new ADR file will be identical to the ADR file on the compiler disk. See the DOS Version 3.3 manual under "EXEC" for more information on EXEC text files.

Appendix B

Creating a Turnkey Disk

It is quite simple to make a turnkey disk that runs a compiled program when the disk is booted. The Applesoft program given in this section BLOADs the RUNTIME library and BRUNs the desired file. Typing this program in as the HELLO program on a disk allows the compiled object code to be executed when the disk is booted. See the DOS Version 3.3 manual for more information on creating turnkey disks.

The following program loads and executes the binary file PROGRAM.OBJ. To use it, change PROGRAM.OBJ to the name of the actual program.

```
10 PRINT CHR$(4) + "BLOAD RUNTIME" + CHR$(13) +  
   CHR$(4) + "BRUN PROGRAM.OBJ"
```

CHR\$(4) is the disk character <CTRL-D>. CHR\$(13) yields a <RETURN>. The PRINT statement executes two disk commands—a BLOAD and a BRUN. The <RETURN> is needed to separate the two commands.

The Applesoft program itself is usually destroyed by the BLOADED RUNTIME, so both the BLOAD and BRUN must be in one PRINT statement. If they were in separate PRINTs, the second PRINT would be destroyed by the BLOAD before it could be executed. This is also why concatenation (+) is used to join the strings. If the strings are not actually concatenated, each substring is output separately, and the same problem occurs.

Appendix C

Notes on Applesoft

The Microsoft Applesoft Compiler is designed to implement the features of Applesoft as closely as possible, so there are very few differences between the interpreter and the compiler. The differences that do exist are explained in Chapter 7. This appendix is included as a further explanation of some of the features and peculiarities of Applesoft. It contains some information about Applesoft statements that is not included in the *Applesoft II BASIC Programming Reference Manual*.

Information is included on the following:

1. TAB and SPC in PRINT statements
2. Reentry of parameters after an error in INPUT
3. Screen wrap-around with DRAW and XDRAW
4. The ONERR GOTO statement
5. FOR/NEXT statements

TAB and SPC Commands

The operation of TAB and SPC is different than might be expected. When used as the last item in a PRINT statement, TAB and SPC act as if they are followed by a semicolon, and suppress the printing of a carriage return.

For example, the following program lines are equivalent:

```
PRINT "SAME" TAB(10) : PRINT "LINE"  
PRINT "SAME" TAB(10); : PRINT "LINE"  
PRINT "SAME" TAB(10) "LINE"
```

All three forms PRINT:

```
SAME    LINE
```

The INPUT Statement

The description of the INPUT statement in the Applesoft manual explains the “?REENTER” message in detail. However, when the “?REENTER” message is displayed, the *entire* INPUT statement is reexecuted. Consider the following response to the statement “INPUT A,B,C”:

2,3,“NOT A NUMBER”

The “?REENTER” message is printed because the string “NOT A NUMBER” cannot be assigned to the numeric variable C. However, Applesoft is requesting the reentry of all three inputs, starting with the value for A. It is not just expecting another single input for C.

Draw and XDRAW Commands

DRAW and XDRAW are HIRES commands for plotting shapes. DRAW and XDRAW require legal X,Y coordinates to specify the starting point for plotting the shape. However, the shape is not necessarily a single point, so it can occupy space on either side of the specified point. As long as the starting point is on the screen, plotting a shape near the edge of the screen does not yield an “?ILLEGAL QUANTITY ERROR.”

If the shape extends past the edge of the screen, the Applesoft DRAW routines wrap the off-screen portion around to the other side. This is not usually a problem, but it can produce abnormal displays.

ONERR GOTO Statement

ONERR GOTO has a problem that causes the interpreter to ignore any statements following an ONERR GOTO statement in a given program line.

The statement PRINT "AND AFTER" in line 10 of the following program is never executed:

```
10 PRINT "BEFORE" : ONERR GOTO 30 : PRINT "AND AFTER"  
20 STOP  
30 PRINT "ERROR"
```

Placing statements after an ONERR GOTO statement is reasonable, since the ONERR GOTO statement does not necessarily transfer control when it is executed. Unfortunately, however, the interpreter *always* ignores them. To avoid any confusion, the compiler handles ONERR GOTO in the same way as the interpreter.

FOR/NEXT Statement

The interpreter's implementation of FOR/NEXT loops includes many special features that remove many of the restrictions normally placed on FOR/NEXT loop nesting. The compiler fully supports all features of Applesoft FOR/NEXT loops.

Appendix D

Runtime Memory Map

Figure 4 shows the default memory configuration for a compiled program. The default can be modified by explicitly specifying the memory addresses at the beginning of compilation. See Chapter 5 for more information.

String variables point into a string area that contains the actual string values. The string values grow downward from the top of memory toward the bottom of the available space. When the strings fill the available memory, the compiler forces garbage collection and frees any unused space. If not enough space is available after garbage collection to store the next string, the compiled code yields an “?OUT OF MEMORY” error.

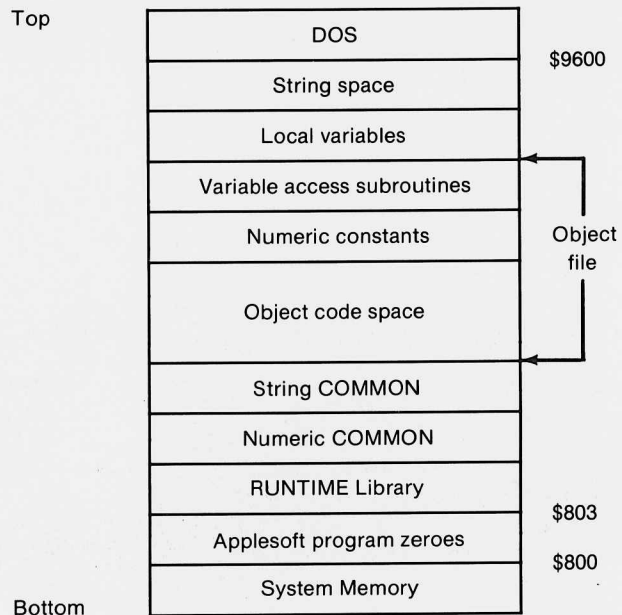


Figure 4. Runtime Memory Map

Appendix E

Zero Page Usage

Compiled programs use routines from the Applesoft interpreter and the RUNTIME library. Both the interpreter routines and the RUNTIME library make extensive use of a portion of memory called the “zero page.” The zero page resides from \$00 to \$FF (decimal 0-255).

Both page zero and page three (hex \$300-\$3FF, or decimal 768-1023) are often used by short machine-language programs also. Compiled code changes the ampersand vector at \$3F5 to allow reexecution of the compiled program, but does not otherwise modify page three.

Zero page usage is confined to the locations that are already used by the Applesoft interpreter. This should eliminate any zero page conflicts with machine-language routines designed to be used with Applesoft.

Compiler Zero Page Usage

Hex address	Usage
0D-0E	Temporaries
10	Temporary
12-13	Address of first DATA string
14-15	Highest location used by library, program, or variables
16	Temporary
50-51	Address temporary
55-56	Temporary
58-5A	JMP to library routine that floats integer accumulator
5B-5D	JMP to code currently handling ONERR
62-66	Temporaries

Applesoft Compiler

Hex address	Usage
67-68	Applesoft beginning of program pointer
69-6A	Beginning of common block-1
6B-6C	End of numeric common
6D-6E	End of string common
6F-70	Bottom of strings-1
71	Input buffer pointer
73-74	Applesoft HIMEM
75-76	Address of object code being executed
77-7A	Temporaries
7B-7C	Beginning of local variables-1
7D-7E	End of local numeric variables
7F-80	End of local string variables
83-84	Temporary
8A-8F	Temporaries
9E-A3	Integer accumulators
AD	Temporary
AF-B0	Address of routine that pops FOR entries from the stack
B1-C8	Applesoft CHRGET, modified, but still functions normally
D9	Temporary
DA-DB	ONERR handling
DC-DD	Temporary
DE	Applesoft error number for error
DF	ONERR handling
F4-F5	ONERR handling
F6-F7	Address of current DATA statement
F8	ONERR handling

Index

- Active REM, 46-48
- ADR, 77, 80-81
 - copying, 80-81
- Ampersand, 28, 91
- Analogy, 12-13
- APCOM, 5, 7, 61-62
- Appendix listing, 4
- Applesoft, 1, 11-12, 14, 45-46
 - pointers, 41-42
 - references, 6
- Applesoft Compiler disk, 5
- Arrays, 33-34, 47, 53

- Backpatching, 62
- BALL program, 7-9
- Binary files, 77-79
- BLOAD, 77-80, 83
- BRUN, 27-28, 52, 54, 56-60,
77-80, 83
- BSAVE, 77-80

- CALL, 28, 74
- Chain with COMMON, 52
- Chapter description, 3-4
- CLEAR, 48, 54, 59-60, 65
 - CHAIN, 48, 54, 59-60
 - COMMON, 48, 54, 59-60
- CLINENUM, 62
- Code generation, 65
- COMMON, 24-26, 39-40, 52-60
 - example, 53, 56-60
- Compilation
 - analogy, 12-13
 - benefits, 1
 - error messages, 69-71
 - explanation, 12-15
 - interpretation comparison,
12-14
 - large programs, 24-26
 - listing, 22
 - process, 61-62
 - statistics, 8-9
 - termination, 24
- Compiled programs
 - execution, 27-29
 - saving and loading, 77-79
 - termination, 28, 34
- Compiletime, 12
- Constants, 23
- Contents of package, 5
- Conversions, 46
- Copying the Applesoft
Compiler, 80-81
- Copyright, ii
- CREATE ADR, 80-81
- CTRL-C, 24, 28, 32, 34

- Debugging, 17-18, 71-72
 - code option, 23-24
- DEF FN, 32
- Defaults, 8
 - memory allocation, 20-21
 - options, 22-24
- DEFCOMMON, 40, 45-48, 52-59
- Definitions, 11-12
- Demonstration run, 7-9
- DIM, 33
- Disk
 - commands, 7-9, 77
 - files, 61-62
 - turnkey, 83
- DRAW, 85-86
- Drive specification, 8

- Editing, 15-18
- END, 40-41
- Error messages, 69-72
 - compilation, 69-71
 - pause option, 22
 - runtime, 71-72
- Expressions, 33, 35

Index

FOR/NEXT, 15, 45, 48, 85, 87
integer, 45, 48

Functions
definition, 32
integer, 49-51

Garbage collection, 43-44
GET, 36
GOSUB, 14, 37
Graphics, 20-21, 50, 72-74, 86

HIMEM, 39, 41, 42
HIRES graphics, 20-21,
72-74, 86

IF/THEN, 35, 51, 70
Ignored statements, 69, 87
Immediate commands, 29
INPUT, 85-86
INTEGER, 45-48
Integer
arithmetic, 22, 45-51
BASIC, 72
constants, 23, 25
FOR/NEXT, 48
math option, 49-51
variables, 45-51
Introduction, 1-6
compilation, 11-15

Keyboard strobe, 34
Keyword, 63

Language compatibility, 18, 31,
78-79
Large programs, 24-26
Lexical analysis, 63
Line numbers, 14, 62, 71

Machine language routines, 39,
74-75
Manual, 3-4
MAXFILES, 39

Memory
map, 89
protection, 20-21
usage, 20-21, 73-74, 90-91
Moving compiled programs,
80-81

NEW, 28, 34, 40-41
Numeric
GET, 36
READ, 36

Object file, 11, 15-16
ONERR GOTO, 23, 37-38, 86-87
Operational differences, 37-44
Options, 19-24

Parsing, 63-64
PASS0, 5, 61
PASS1, 5, 61-62
PASS2, 5, 8, 61-62
Pause on errors, 22
PEEK, 74
POKE, 55, 73-75
PRINT, 18, 85-87
Printers, 9
Program
development 15-16
security, 2
self-modifying, 75
systems, 52-53
Prompts, 7-9

READ, 36
RECOGNIZED, 48
REM, 46-48
Requirements, 2, 5
RESET, 24, 28, 34, 41
RESUME, 23-24, 38
RUN, 27, 40, 42, 77-78
Runtime
definition, 12
error messages, 71-72
library, 12, 14, 20-21, 27,
68, 78-79, 83, 91-92
Runtime license requirements, 5

- Shape tables, 73
- Slot specification, 8
- Source file, 11, 15-16
- SPC, 85-86
- Stack, 37-38
- Statements
 - added, 45
 - ignored, 31, 69, 81
 - integer, 46-48
- STOP, 40-41
- STR\$, 35
- String
 - chaining, 52, 54
 - expressions, 35
 - garbage collection, 43-44
 - IF/THEN, 35
 - operations, 43-44
 - space, 21
 - storage, 54-56, 89
- Symbol table, 25-26, 62
- Syntax analysis, 63-64
- System requirements, 2

- TAB, 85-86
- Token, 63
- Translation, 12-14
- Triples, 64
- Turnkey disks, 83

- USECOMMON, 40, 45, 52-53, 56-60
- Utilities, 80-81

- Variables
 - accessing, 66-67
 - clearing, 40
 - FOR/NEXT, 48
 - list, 14

- XDRAW, 85-86

- Zero page, 41, 91-92



10700 Northup Way, Bellevue, WA 98004

Software Problem Report

Name _____

Street _____

City _____ State _____ Zip _____

Phone _____ Date _____

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

_____ Software Problem

_____ Documentation Problem
(Document # _____)

_____ Software Enhancement

_____ Other

Software Description

Microsoft Product _____

Rev. _____ Registration # _____

Operating System _____

Rev. _____ Supplier _____

Other Software Used _____

Rev. _____ Supplier _____

Hardware Description

Manufacturer _____ CPU _____ Memory _____ KB

Disk Size _____ " Density: _____ Sides: _____

Single _____ Single _____

Double _____ Double _____

Peripherals _____

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support _____

Date Received _____

Routing Code _____

Date Resolved _____

Report Number _____

Action Taken: